VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA PODNIKATELSKÁ

FACULTY OF BUSINESS AND MANAGEMENT

ÚSTAV INFORMATIKY

INSTITUTE OF INFORMATICS

VÝVOJ GENERÁTORU SOUBORŮ

FILE GENERATOR DEVELOPMENT

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

AUTOR PRÁCE — Bc. Šimon Procházka

AUTHOR

VEDOUCÍ PRÁCE — Ing. Petr Dydowicz, Ph.D.

SUPERVISOR

BRNO 2021

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA PODNIKATELSKÁ

Zadání diplomové práce

Ústav:	Ústav informatiky
Student:	**Bc. Šimon Procházka**
Studijní program:	Systémové inženýrství a informatika
Studijní obor:	Informační management
Vedoucí práce:	**Ing. Petr Dydowicz, Ph.D.**
Akademický rok:	2020/21

Ředitel ústavu Vám v souladu se zákonem č. 111/1998 Sb., o vysokých školách ve znění pozdějších předpisů a se Studijním a zkušebním řádem VUT v Brně zadává diplomovou práci s názvem:

Vývoj generátoru souborů

Charakteristika problematiky úkolu:

Úvod
Vymezení problému a cíle práce
Teoretická východiska práce
Analýza problému a současné situace
Vlastní návrh řešení, přínos práce
Závěr
Seznam použité literatury

Cíle, kterých má být dosaženo:

Vývoj generátoru souborů v prostředí příkazové řádky, který umožní popsat databázové struktury Microsoft SQL Serveru. Z prostředí příkazové řádky by také mělo být možné vygenerovat soubory ve formě zdrojového kódu v jazyce C# a TypeScript.

Základní literární prameny:

BASL, J. a R. BLAŽÍČEK. Podnikové informační systémy. Podnik v informační společnosti. Praha: Grada, 2008. 283 s. ISBN 978-80-247-2279-5.

MOLNÁR, Z. Automatizované informační systémy. Praha: Strojní fakulta ČVUT, 2000. 126 s. ISBN 80-01-02269-2.

MOLNÁR, Z. Efektivnost informačních systémů. Praha: Grada Publishing, 2000. 142 s. ISBN 80-716--410-X.

ŘEPA, V. Analýza a návrh informačních systémů. Praha: Ekopress, 1999. 403 s. ISBN 80-86119-- 3-0.

SODOMKA, P. a H. KLČOVÁ. Informační systémy v podnikové praxi. Brno: Computer Press, 2010. 501 s. ISBN 978-80-251-2878-7.

Termín odevzdání diplomové práce je stanoven časovým plánem akademického roku 2020/21

V Brně dne 28.2.2021

L. S.

Mgr. Veronika Novotná, Ph.D.	doc. Ing. Vojtěch Bartoš, Ph.D.
ředitel	děkan

Abstrakt

Diplomová práce je zaměřena na vývoj generátoru souborů, který bude sloužit jako podpůrný nástroj vývoje webových aplikací ve společnosti LOGEX Solution Center s.r.o. Tato aplikace byla vytvořena za využití prostředí Node.js v jazyce TypeScript.

Klíčová Slova

Generátor souborů, Node.js, JavaScript, TypeScript, LOGEX

Abstract

The topic of this master's thesis is the development of a file generator which will be used as a support tool for web application development in LOGEX Solution Center s.r.o. The application was written in TypeScript using the Node.js JavaScript runtime.

Key words

File Generator, Node.js, Javascript, TypeScript, LOGEX

Bibliografické citace

PROCHÁZKA, Šimon. *Vývoj generátoru souborů* [online]. Brno, 2021 [cit. 2021-04-24]. Dostupné z: https://www.vutbr.cz/studenti/zav-prace/detail/135325. Diplomová práce. Vysoké učení technické v Brně, Fakulta podnikatelská, Ústav informatiky. Vedoucí práce Petr Dydowicz.

Čestné prohlášení

Prohlašuji, že předložená diplomová práce je původní a zpracoval jsem ji samostatně. Prohlašuji, že citace použitých pramenů je úplná, že jsem ve své práci neporušil autorská práva (ve smyslu Zákona č. 121/2000 Sb., o právu autorském a o právech souvisejících s právem autorským).

V Brně dne 16. 5. 2021

……...

podpis studenta

Poděkování

Rád bych touto formou poděkoval vedoucímu mé diplomové práce Ing. Petru Dydowiczi, PhD. za vedení diplomové práce a čas, který mi věnoval. Dále bych rád poděkoval společnosti LOGEX Solution center s.r.o. a jejím zaměstnancům za aktivní přístup a pomoc při zpracovávání práce.

OBSAH

Úvod

Vývoj webových aplikací v posledních letech ušel obrovský kus cesty. Popularita webu dala za vznik novým technologiím, které umožňují skriptování na straně serveru skrze jazyky, pro které to dříve bylo nemyslitelné.

V současné době zažívají velký boom komunikační aplikace jako Discord, Slack či Microsoft Teams. Přestože je jejich popularita velmi spjatá s koronavirovou situací ve světě, která přesunula pracoviště do domácností, všechny tyto aplikace mají z technologického hlediska společnou jednu věc, a to JavaScript.

Od roku 2009 přišla na svět technologie Node.js, která umožnila všechny tyto aplikace díky možnosti spouštět JavaScriptový kód na počítači, a přispěla k tomu, že se již nejedná o jazyk určený pouze pro vývoj webových prvků, se kterými může uživatel pracovat.

Po možnostech, které přinesl Node, přišel na svět Electron. Framework umožňuje vytvářet desktopové aplikace v JavaScriptu, a výše zmíněné komunikační technologie mají společný právě tento framework. Kromě komunikačních aplikací do této kategorie můžeme také zařadit streamovací platformu Spotify či vývojové prostředí Visual Studio Code.

Myšlenka využití jednoho jazyka pro všechny prostředí je velmi zajímavá pro podnikatele, kteří nemusí zaměstnávat několik různých specialistů na různé technologie, ale stačí jim pro všechny cílené zařízení jediný jazyk, a touto vizí nižších nákladů se snaží vytvářet nové aplikace na těchto platformách.

Věřím, že JavaScript a technologie s ním spojené zde budou ještě dlouho, a i z toho důvodu jsem se rozhodl zaměřit svoji diplomovou práci právě na tyto technologie, konkrétně tak jejich použití na straně serveru

VYMEZENÍ PROBLÉMU A CÍLE PRÁCE

Cílem diplomové práce je návrh a vývoj aplikace pro firmu LOGEX Solution Center s.r.o., která generuje zdrojový kód pro volání databázových procedur. Společnost již má aplikaci, která slouží pro tyto účely, je však přístupná pouze skrze webový prohlížeč a vývojář musí ručně kopírovat a vytvářet soubory.

V práci se řídím požadavky, které definovaly zaměstnanci firmy, se kterými jsem také konzultoval možné přístupy.

Mezi funkční požadavky patří automatické generování souborů v počítači a aplikace musí být spustitelná skrze příkazovou řádku. Volané databázové procedury jsou napsány v Transact-SQL.

Technologické požadavky se zabývají prostředím, tedy že aplikace musí napsána v jazyce TypeScript a využívat Node.js pro spouštění kódu mimo webový prohlížeč.

Mezi další požadavky patří přizpůsobení aplikace uživateli. Tato přizpůsobení jsou ve formě dodatečných parametrů, které je možné definovat při spuštění programu. Příkladem může být možnost definovat databázi, ze které chceme procedury volat. Celkový počet požadovaných konfigurací je 13.

V analytické části jsem metodou průzkumu trhu zjistil, jaké nástroje generující zdrojový kód existují, a dále jsem je využil jako inspiraci při samotném návrhu a implementaci programu.

V návrhové části jsem použil metody pro návrh designu, konkrétně tak konzistentní strukturu a volání metod aplikace, abych zaručil příjemnou zkušenost uživatele s používáním aplikace.

1 TEORETICKÁ VÝCHODISKA PRÁCE

Pro přiblížení problematiky vývoje příkazového řádku v této kapitole popisuji teoretická východiska, na jejichž základě je práce postavena.

1.1 Příkazový řádek

Příkazový řádek je textové rozhraní pro spouštění programů. V obecné rovině se bavíme o příkazovém řádku, pro lepší technologické pochopení je však nutné rozlišovat dva termíny – terminál a shell. Těmito termíny se více zabývám v dalších podkapitolách. Nakonec popisuji tři známé shelly v systému Windows, jejichž prostředí moje aplikace podporuje.

1.1.1 Terminál

Terminál je jednoduché vstupní a výstupní zařízení. Jedná se o zařízení, které není schopné provádět žádnou logiku. Slouží pouze k zadání vstupu uživatelem, předání vstupu dalším zařízením, a zobrazení výstupu uživateli. (1)

Mezi první terminály lze zařadit dálnopis. Na rozdíl od moderních obrazovek se jednalo o zařízení, které je podobné psacímu stroji. Vše, co napsal uživatel, se objevilo na dodaném papíru. V případě odpovědi pak stroj sám vyťukal na stejný papír. Takový případ terminálu je možné najít na obrázku 1. (1)

Obrázek 1: Dálnopis Telex (Zdroj: 2)

V případě terminálu se tedy jedná o jednoduché vstupně-výstupní zařízení, které není schopné složitější logiky. V současné době jsou tímto termínem označovány spíše prostředí příkazových řádků. Moderní terminál v prostředí dodaném instalací software *Git* je možné vidět na obrázku 2. (1)

Obrázek 2: Terminál spuštěný ve Windows 10. (Zdroj: vlastní zpracování)

1.1.2 Shell

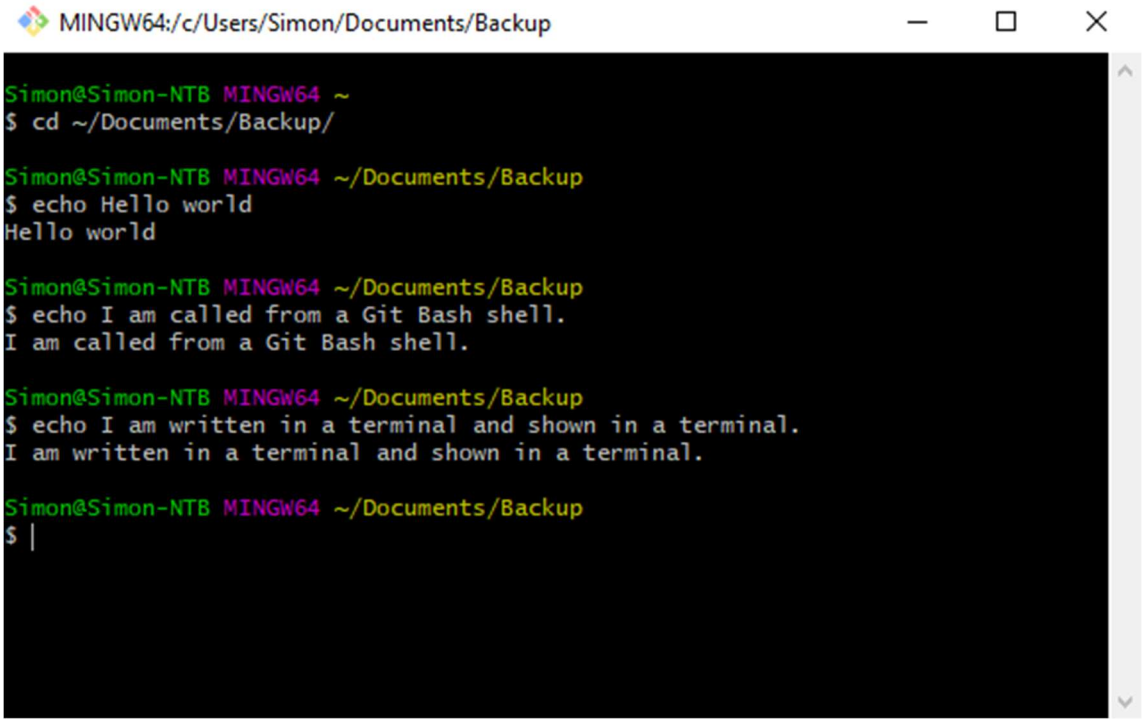
Vzhledem k tomu, že terminál není schopný jakékoliv složitější logiky, některé programy jsou automaticky spuštěny při jeho startu. Takové programy zajišťují veškerou funkcionalitu a nazývají se *Shell*. (1)

Zatímco na obrázku 2 vidíme prázdný terminál, v případě zavolání některých funkcí správnou funkci zajistí program *Shell*. Při každém zadání vstupu v terminálu se tento vstup odešle do *Shellu*, který vygeneruje požadované návratové hodnoty, a vrátí je terminálu k zobrazení. (1)

Například příkaz *cd* (zkráceně pro *change directory*, tedy změna adresáře), který umožňuje v terminálu navigovat mezi různými složkami. V případě zadání názvu složky změní současný pracovní adresář na požadovaný. Díky tomu jsme schopni v rámci

terminálu pracovat s celým souborovým systémem, aniž bychom k tomu potřebovali prohlížeč a myš. (1)

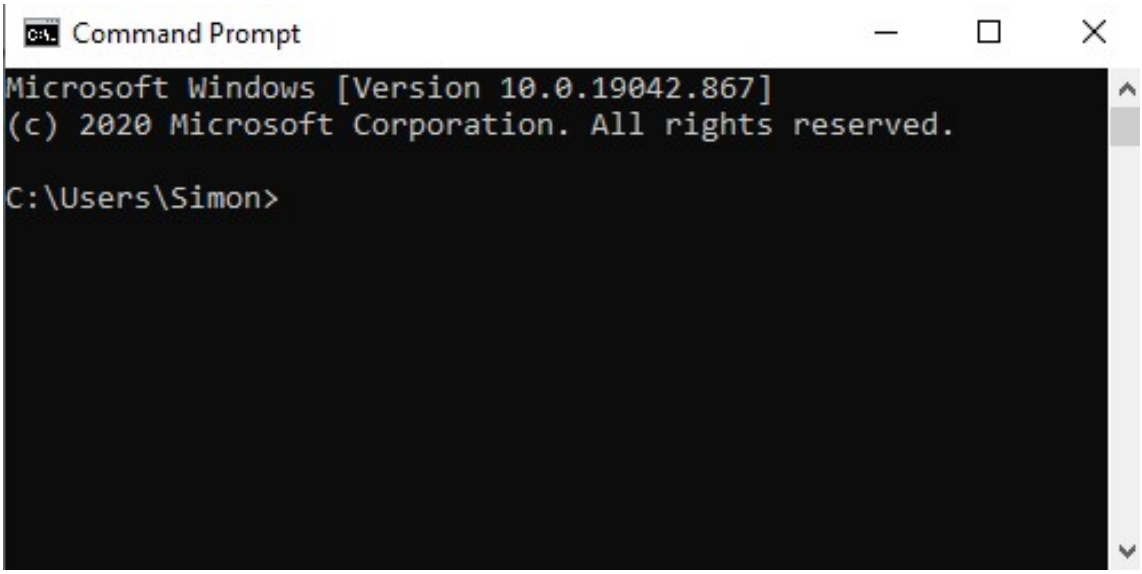
Na obrázku 3 je možné vidět příklad volání příkazů uvnitř *Shellu*. Veškerý vstup, který jsem jako uživatel napsal, předchází znak $. V prvním volání jsem využil funkci *change direktory*, která změnila současný pracovní adresář. Veškeré další příkazy využívají *Shell* funkci *echo*, která vypíše do terminálu požadovaný text.



Obrázek 3: Volání shell příkazů za využití terminálu (Zdroj: vlastní zpracování)

1.1.3 Command Prompt

Command Prompt neboli cmd.exe je první shell zabudovaný do Windows pro automatizaci rutinních úkolů, jako například řízení uživatelských účtů či noční zálohy. Tuto funkcionalitu umožňuje skrze dávkové soubory .bat. Jedná se o zabudovaný software v systému Windows, který umožňuje komunikaci přímo s operačním systémem (3). Využití tohoto shellu je možné vidět na obrázku 4.



Obrázek 4: cmd.exe v systému Windows (Zdroj: vlastní zpracování)

1.1.4 Bash

Další shell je možné znát pod názvem Bash. Jedná se o UNIX shell, který byl vyvinut takovým způsobem, aby odpovídal normě IEEE 1003.1-2017. Tato norma definuje *Shell Command Language* (4) a má mnoho implementací, mezi které patří například *dash* (5) či *ksh* (6). Ve všech zmíněných případech se bavíme o implementacích *shellu*, což značí zkratka *sh* na konci každého jména. Popularitu si Bash zajistil zejména tím, že slučuje užitečné funkce z různých shellů, mezi které patří i dříve zmíněny *ksh*. (7)

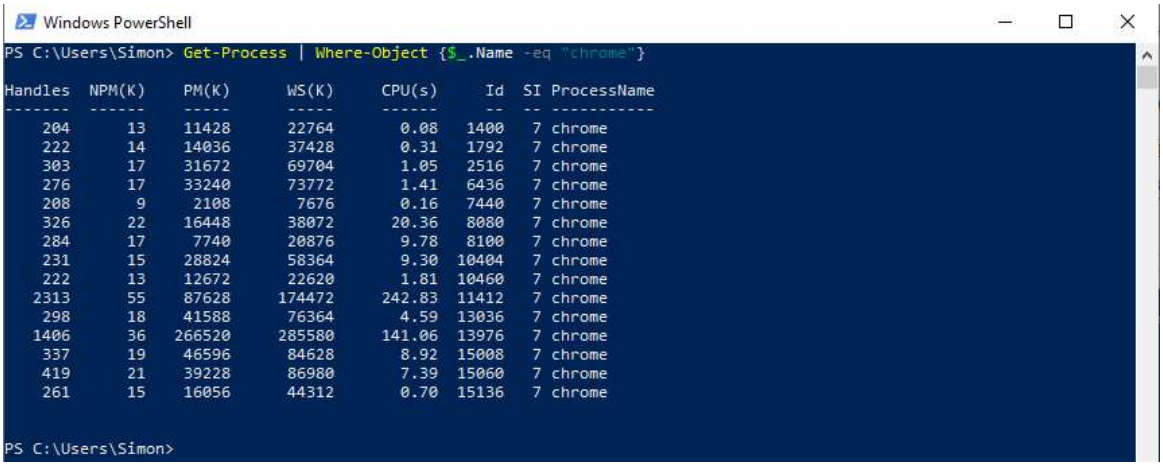
Všechny zmíněné shelly v této podkapitole jsou součástí UNIX ekosystému. Pro používání *Bash* ve Windows je nutné využít některou z dostupných nástaveb. Jedna implementace je možná vidět na obrázku 3 a jedná se o *Git Bash*. (7)

1.1.5 Powershell

Posledním shellem, kterým se zabývám v této kapitole, je *PowerShell*. PowerShell umožnuje automatizaci úloh pro různé platformy a správu konfigurace. V novějších verzích Windows je také automaticky nainstalovaný a je možné jím úplně nahradit dříve zmíněný Command Prompt. (8)

Na rozdíl od cmd.exe, který využívá terminál, PowerShell je postavený na jiných technologiích, konkrétně .NET Common Language Runtime. Díky tomu je schopný pracovat nejen s textem, ale také s objekty. Má také vlastní skriptovací jazyk. (8)

Na obrázku 5 je možné vidět ukázku Windows PowerShell, kde využívám *přesměrování* výsledků za využití objektu. Tento příkaz by byl neplatný v cmd.exe, protože využívá skriptovacího jazyka vestavěného do *PowerShellu.* (8)



Obrázek 5: Windows PowerShell (Zdroj: vlastní zpracování)

1.2 Technologie pro vývoj aplikací v příkazové řádce

V následující kapitole se zabývám technologiemi, které umožňují vývoj aplikací uvnitř příkazové řádky. Takovou aplikaci je možné psát v jakémkoliv jazyce, který umožňuje vytvoření spustitelných souborů v počítači. Typicky se tak jedná o jakýkoliv jazyk, který umožňuje skriptování na straně serveru.

Vzhledem k tomu, že zadání mé práce obsahuje jazyk TypeScript, což je skriptovací jazyk na straně klienta, není možné bez nástavby tento úkol splnit. Z toho důvodu se v této kapitole zabývám technologiemi, které je nutné využít pro tyto účely, včetně samotného jazyka.

1.2.1 JavaScript

První technologií, kterou se zabývám v této podkapitole, je JavaScript. Jedná se o interpretovaný kompilovaný jazyk, který je známý zejména za jeho využití ve webových stránkách. Díky technologiím jako Node.js či Deno je však možné jej využít i v jiných prostředích. (9)

Norma, na které je JavaScript postavený, se jmenuje ECMAScript (10), v současné době konkrétně její 11. edici. Jedním z nejčastějších problémů, na které můžeme při vývoji JavaScriptu narazit, je nekompatibilita jeho funkcí se staršími

prohlížeči a jinými technologiemi. Pro účely mapovaní těchto problémů existuje mnoho nástrojů (11).

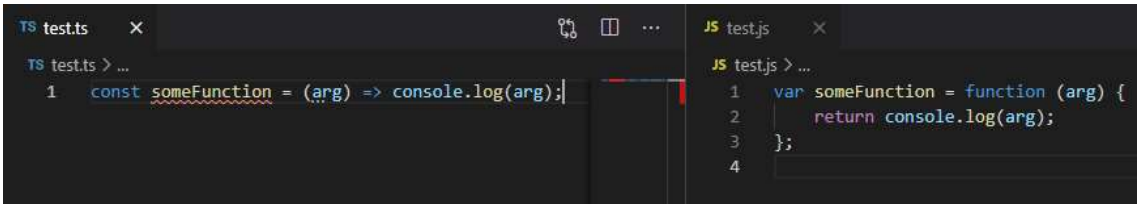
Specifikace JavaScriptu umožňuje nahradit nativní metody vlastním kódem. V praxi to znamená, že takové metody můžeme úplně přepsat v konkrétních případech – například pokud metoda. Příklad nahrazení nativní metody *isArray* je možné vidět na obrázku 6, kde v první případě je možné vidět, že funkce správně rozpozná, zda argument zaslaný do funkce je pole či ne. Avšak po mém zásahu do nativní funkce již není schopné tuto funkci k tomuto účelu použít, neboť vždy vrátí hodnotu *true*. (9; 12)

```
> Array.isArray("a")
< false
> Array.isArray(["a"])
< true
> Array.isArray = (arg) => true
< (arg) => true
> Array.isArray("a")
< true
```

Obrázek 6: Ukázka nahrazení nativní metody (Zdroj: vlastní zpracování)

Jak jsem již zmínil, tato funkcionalita se používá nejvíce v případě, kdy se snažíme přidat funkcionalitu do starších verzí prohlížečů, ve kterých tato funkce zatím neexistuje. V takovém případě je možné vyhledávat pod termínem *polyfill*. (12)

Podobnou funkcionalitu můžeme najít také pod jiným názvem, a to *transpilace*. Transpilace slouží k velmi podobnému účelu – nahradit nepodporovaný kód jiným, který provede stejnou funkcionalitu, ale podporuje jej zvolené prostředí. Jak je možné vidět na obrázku 7, levá strana používá funkci definovanou skrze šipku, která podporovaná není, zatímco pravá strana je jednoduchý JavaScript, který funguje všude. (13)

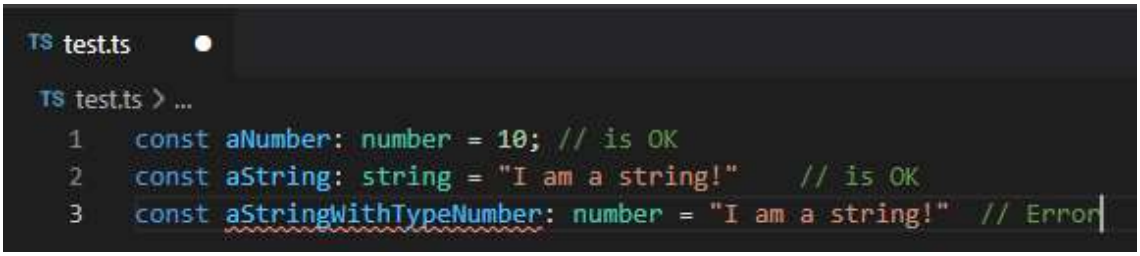


Obrázek 7: Příklad transpilace (Zdroj: vlastní zpracování)

1.2.2 TypeScript

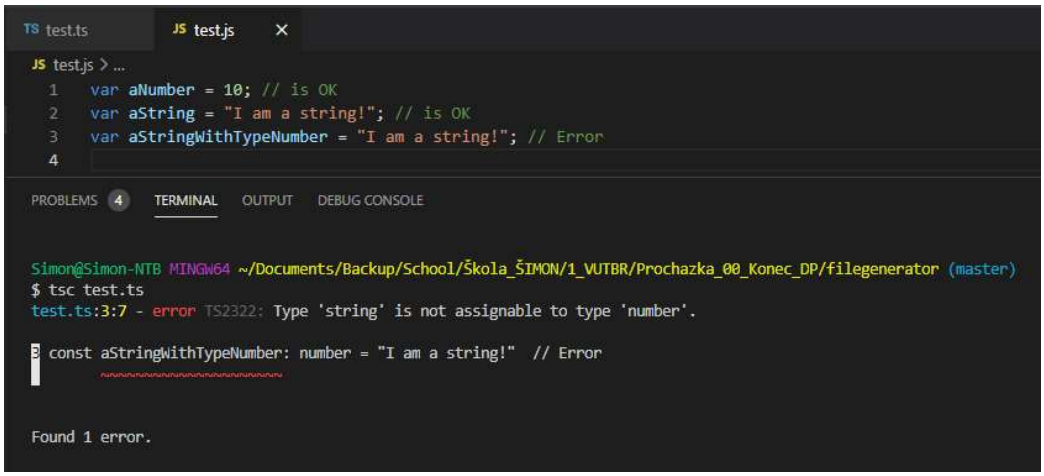
Na obrázku 7 je kromě transpilace vidět také soubor s příponou *.ts*. Tento soubor označuje jazyk TypeScript, který je hlavním jazykem použitý v diplomové práci. Vzhledem k tomu, že není možné tento jazyk spustit bez transpilace do JavaScriptu, rozhodl jsem se jej umístit na druhé místo použitých technologií (13).

TypeScript je open-source jazyk, který je postavený na JavaScriptu. Veškerý kód validní v JavaScriptu je také validní v TypeScriptu. Jak již název napovídá, jedná se o typovaný JavaScript, díky čemuž pomáhá vývojářům v přehlednosti zdrojového kódu. Na rozdíl od jiných typovaných jazyků jako Java či C#, Tento jazyk nevynucuje používání typů a nepotřebuje je ke správné funkci. (13)



Obrázek 8: Příklad zdrojového kódu v jazyce TypeScript (Zdroj: vlastni zpracování)

Na obrázku 8 je možné vidět příklad zdrojového kódu v TypeScriptu. První dva řádky definují typované konstanty, které neobsahují žádné chyby. Třetí řádek se však snaží vložit řetězcovou hodnotu do číselného typu a konstanta je červeně podtržená, což značí chybu. Tato chyba se projeví také při transpilaci do JavaScriptu, jak je možné vidět na obrázku 9.



Obrázek 9: Ukázka transpilace TypeScriptu do JavaScriptu (Zdroj: vlastní zpracování)

Transpilaci TypeScript umožňuje skrze příkaz *tsc* (13), která transpiluje TypeScriptový kód do JavaScriptového. Pokud definované typy nejsou v pořádku, příkaz vyhodí do terminálu chybovou hlášku. Přestože se při transpilaci objevila chyba, transpilace proběhne v pořádku, a pokud se pokusíme kód spustit, bude fungovat v pořádku. Tento případ je možné vidět na obrázku 10.

```
> var aNumber = 10; // is OK
  var aString = "I am a string!"; // is OK
  var aStringWithTypeNumber = "I am a string!"; // Error
< undefined
> aStringWithTypeNumber
< "I am a string!"
```

Obrázek 10: Spuštění transpilovaného kódu (Zdroj: vlastní zpracování)

Na obrázku je možné vidět, že transpilovaný kód, přestože vyhodil chybové hlášky, je funkční, a proměnná *aStringWithTypeNumber* obsahuje definovanou hodnotu. Je to z toho důvodu, že TypeScript je pouze tak silným nástrojem, jak kvalitní je jeho použití. Přestože se objevily chyby, TypeScript dokončil kompilaci a pouze vývojáře informoval chybovou hláškou, že tento problém nastal. Jedná se tak spíše o informativní nástavbu JavaScriptu, která se snaží navést vývojáře správným směrem, ale nezablokuje mu cestu k výsledku.

1.2.3 Node.js

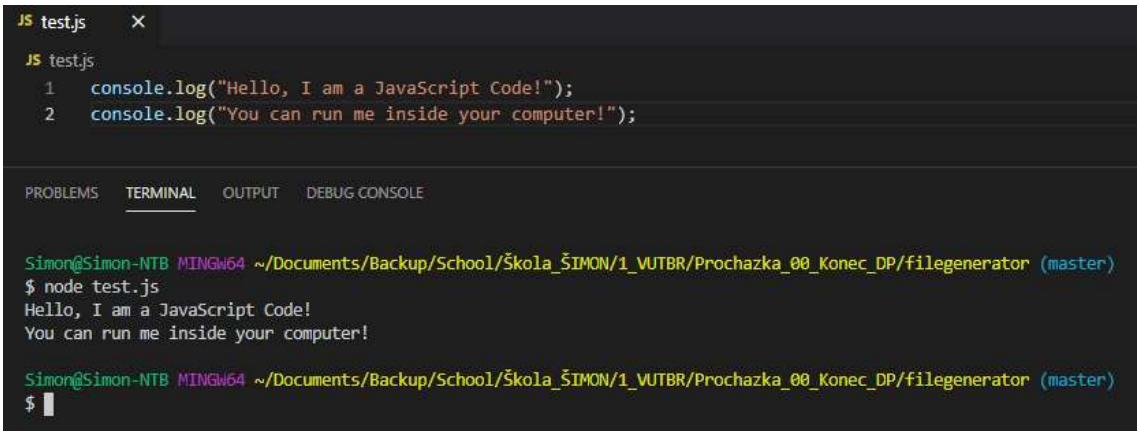
V rámci diplomové práce se zabývám vytvářením aplikace, která je spustitelná z prostředí příkazového řádku. Jak jsem již dříve zmínil, TypeScript je nutné transpilovat do JavaScriptu, a JavaScript není možné spustit v jiném prostředí než v prohlížeči bez dalších nástrojů. Prvním z těchto nástrojů je Node.js.

Node.js je prostředí, které umožňuje spouštět JavaScript mimo webové prostředí. Primární účel je tvorba serverových aplikací a umožňuje jazyk používat ke skriptování na straně serveru. (14)

Díky tomuto nástroji je tak možné spustit JavaScript na počítači, a vývojář vůbec nepotřebuje prohlížeč. Aby to bylo možné, je nutné si nejprve nainstalovat Node.js

aplikaci. V příkazovém řádku poté můžeme použít příkaz *node*, který provede kód, který tomuto příkazu zašleme. (14)

Na obrázku 11 je možné vidět jednoduché zavolání JavaScriptového kódu pouze za využití terminálu.



Obrázek 11: Provedení JavaScriptového kódu mimo prohlížeč (Zdroj: vlastní zpracování)

1.2.4 Deno

Deno je další technologií, která umožňuje spouštění JavaScriptového mimo prohlížeč. Přestože ho v práci nepoužívám, věřím, že je vhodné jej zmínit, protože se jedná o další populární nástroj k dosáhnutí stejného výsledku.

Projekt Deno založil Ryan Dahl, který dříve pracoval na Node.js. Důvodem ke vzniku tohoto projektu bylo mnoho věcí, které autor považoval za problematické u Node. Jedním z těchto problémů je udržování Node.js, který je sice open-source, jeho správcem je však soukromá společnost Joyent, což dalo za vznik mnoha spekulacím ohledně nestrannosti Node.js a zda se společnost nebude snažit projekt privatizovat. Privatizace tak populárního nástroje by totiž mohla mít devastující účinky pro velké množství software. (15; 16)

Vztah mezi těmito dvěma projekty se stane jasnějším, pokud bychom seřadili písmena ve slově *node* vzestupně. V takovém případě vznikne slovo *deno*, tedy název druhého nástroje. Autor se tím snaží říct, že se jedná o verzi node, která má svoje priority seřazené. (16)

1.2.5 Node package manager

Dalším nástrojem, který v práci hojně využívám, je Node package manager, respektive *npmjs*. Jedná se o největší software registr na světě, který v současné době má více než 1.6 milionu software balíčků, které si může jakýkoliv vývojář jednoduše stáhnout bez registrace. (17)

Správce balíčků je automaticky nainstalovaný společně s Node.js a instalaci balíčků zajišťuje skrze příkaz *npm install*. Tento příkaz nainstaluje všechny balíčky, které jsou součástí konfiguračního souboru *package.json*. Pokud chceme nainstalovat pouze jediný balíček, můžeme za příkaz dodat jméno požadovaného balíčku, a nainstaluje se pouze námi specifikovaný balíček. (17)

V diplomové práci tento balíček hojně využívám k získání knihoven třetích stran, které mám dále sepsány v kapitole 1.4. Kromě využívání jiných balíčků je diplomová práce také volně dostupná na tomto registru, je tedy možné jej nainstalovat skrze tento software. (17)

1.2.6 Yarn

Yarn je další z rodiny správců balíčků. Podobně jako *npmjs*, tento správce instaluje knihovny třetích stran skrze soubor *package.json*. Pomocí příkazu *yarn install* nainstalujeme veškeré závislosti našeho projektu. Ve své diplomové práci se zabývám převážně *npmjs*, tento nástroj však je možné využít ke stejnému cíli. (18)

1.3 Synchronní a asynchronní programování

Synchronní a asynchronní programování je nedílnou součástí vývoje softwaru. JavaScript je v tomto ohledu komplikovanější, jedná se totiž o jedno-vláknový programovací jazyk. To znamená, že není možné provádět více operací zaráz. Je však možné zde využít iluzi asynchronního programování. (19)

Pro kvalitnější vysvětlení asynchronního programování v JavaScriptu využiji metaforu pro vaření, jedná se totiž o stejný princip. V případě, že jsme v kuchyni a vaříme jídlo, nejsme schopní dělat více věcí zaráz – nemůžeme například zároveň krájet cibuli a česnek. Víme však, že se cibule smaží déle. Z toho důvodu nejprve nakrájíme cibuli, dáme ji do pánve, a zatímco se cibule smaží, nakrájíme česnek, a nakonec ho také přidáme

od pánve. Tak dosáhneme toho, že jsme schopní dělat více věcí zaráz, přestože tomu tak úplně není.

Pokud bychom tuto metaforu zjednodušili, jedná se o sérii úkonů, které děláme asynchronně. Nejprve začneme jeden úkon, a dokud si nebude vyžadovat znovu naši pozornost, zaměříme se na jiný. Na stejném principu funguje i asynchronní programování. (19)

Tento princip je velmi důležitý zvlášť v případě, že pracujeme se soubory v počítači, což je případ i mé diplomové práce. Můžeme například začít čtení více souborů naráz, avšak musíme počkat, až všechny tyto soubory budou přečtené, abychom mohli dělat závěry.

1.3.1 Synchronní programování

Jak již název napovídá, synchronní programování je přímočarý způsob vykonávání úkolů. Řádky v kódu se provádí tak, jak jdou za sebou. Na obrázku 12 je možné vidět jednoduchý kód, který vypisuje do konzole. Jak je možné vidět, řádky se provedou v pořadí, v jakém byly definovány.

```
> console.log("first log");
  console.log("second log");
  console.log("third log");

  first log

  second log

  third log
```

Obrázek 12: Příklad synchronního programování (Zdroj: vlastní zpracování)

1.3.2 Asynchronní programování

Asynchronní programování a důvod pro jeho využití jsem již popsal v úvodu této kapitoly. V této části se zaměřím primárně na implementaci kódu takovým způsobem, aby byl asynchronní. JavaScript využívá celkem tři konstrukce – task queue, event loop a WebAPI. Specifickou kombinací některých z nich umožňuje vývojářům dosáhnout asynchronního programování. (20)

Task queue definuje pořadí, ve kterém se mají vykonat jednotlivé úkony. V případě příkladu na obrázku 12 se tak jedná o frontu, která obsahuje tři příkazy, respektive tři výpisy do konzole. (20)

V případě event loop se bavíme o nekonečné smyčce, do které putují úkony z task queue a reálně se provedou. Každá definovaná funkce se provede v jedné smyčce, a po jejím ukončení se do smyčky přidají další funkce z fronty, dokud není prázdná. (20)

Poslední konstrukcí je WebAPI, a je velmi specifická v tom, že není součástí samotného JavaScriptu, nýbrž prostředí, ve kterém se využívá. Například internetové prohlížeče mají vlastní implementace některých metod, což je mimo jiné důvod, že se stejný kód chová jinak v různých prohlížečích, jak jsem zmínil v dřívějších kapitolách. (21)

Vzhledem k tomu, že je součástí prostředí, nemusí nutně být napsané v JavaScriptu, ale v jiných jazycích, ve kterých jsou prostředí vytvořena. Tyto jazyky mohou být více-vláknové a mohou tak provádět více operací zaráz. Je tak možné některé operace úplně oddělit od běhu našeho programu a provádět je například v prohlížeči. (20; 21)

Program je nakonec prováděn tak, že nejprve se volání přidá do task queue, poté vstoupí do smyčky, která samotné volání provádí. V době provádění těchto volání se přidávají do call stack. V případě, že využíváme některé metody prostředí, se kterým pracujeme, je možné tato volání úplně vypustit a nechat je provést například prohlížečem. (20; 21)

```
> function logItemsAndReturnOne() {
      console.log("first log");
      setTimeout(() => console.log("second log"));
      console.log("third log");
      return 1;
  }
< undefined
> logItemsAndReturnOne()
  first log
  third log
< 1
  second log
```

Obrázek 13: Příklad asynchronního programování (Zdroj: vlastní zpracování)

Příklad asynchronního volání je možné vidět na obrázku 13, kde jsou výpisy v konzoli ve stejném pořadí jako na posledním příkladu. Rozdílem však je, že druhý výpis do konzole je obalený funkcí *setTimeout*. Tato funkce je WebAPI, jedná se tedy o funkci, kterou nepodporuje jazyk, nýbrž prostředí. Po zavolání této funkce luze vidět, že nejprve se provede první a poslední výpis, poté se vrátí hodnota z funkce, a nakonec se provede druhý výpis v pořadí.

1.4 Další nástroje pro vývoj aplikací v příkazové řádce

Snad každý vývojář ve své kariéře využil nějakou knihovnu třetí strany. Moje diplomová práce není výjimkou, a proto se v této kapitole zabývám dalšími nástroji, které v práci využívám. Nejprve definuji tři termíny, které je vhodné pochopit před prací s dalšími nástroji, dále následuje výpis všech použitých technologií.

1.4.1 Knihovna

Při využívání nástrojů třetích stran v našem projektu se bavíme o balíčcích nebo knihovnách. Tento termín označuje funkcionalitu, kterou můžeme využívat v našem zdrojovém kódu, avšak nepatří přímo nám. Příkladem knihovny může být například jeden z nejznámějších balíčků *jQuery*. Balíčky je možné jednoduše nainstalovat skrze *npmjs*. (22)

1.4.2 Framework

V případě frameworku se v informatice bavíme o struktuře, která definuje architekturu našeho řešení. Tato struktura může být ve formě jednoho balíčku, ale může také být definována větším množstvím vzájemně provázaných balíčků. Příkladem mohou být nejznámější webové technologie *Angular*, *React* či *Vue*. Stejně jako u knihovny i framework je možné nainstalovat skrze *npmjs*. (22)

1.4.3 API

API neboli rozhraní pro programování aplikací (z anglického *Application Programming Interface*) definuje přístup k funkcím knihoven či frameworků. U mnoha dokumentací můžeme najít záložku *API Reference*, která definuje popis a využití

jednotlivých funkcí balíčků. Ve zkratce se jedná o definované názvy a odkazy na reálnou implementaci, funkce API je tak spíše informativní. (22)

1.4.4 Yargs

Nyní, když známe všechny důležité termíny pro práci s balíčky, je možné popisovat jednotlivé balíčky. Prvním z těchto balíčků je *yargs*, který slouží pro usnadnění práce s vývojem interaktivních programů v prostředí příkazového řádku. (23)

Tento balíček umožňuje vývojářům usnadnit práci s definováním příkazů a parametrů, což je velmi užitečné zvlášť pro tuto diplomovou práci. Jedním z požadavků společnosti je možnost konfigurace a přizpůsobení konkrétních příkazů. Balíček umožňuje definovat příkazy skrze funkci *command* a ke každému příkazu namapovat konkrétní možnosti funkcí *option*. Tento balíček je v práci hojně využitý při zavolání aplikace, kde slouží k získání všech příkazů a parametrů. (23)

1.4.5 ts-node

Dalším balíčkem, který v práci využívám, je ts-node. Tento balíček je nutný z toho důvodu, že práce má být napsaná v jazyce TypeScript. V kapitole 1.2 jsem zmínil, že Node.js umožňuje práci s JavaScriptem mimo webový prohlížeč, a že TypeScript je nutné transpilovat, aby bylo možné jej použít. Abych nemusel pracně po každé změně manuálně volat kompilaci, využívám tento balíček k tomu, abych mohl rovnou volat TypeScriptový kód. (24)

1.4.6 fs-extra

Cíle práce definují, že aplikace musí vygenerovat soubory v počítači. Práci se souborovým systémem jsem se rozhodl zajistit balíčkem *fs-extra*, který rozšiřuje defaultní modul o užitečné funkcionality a rychlost. Jednou z těchto funkcionalit je rozšíření modulu o typy pro TypeScript a je tak možné využít jeho plný potenciál. (25)

1.4.7 figlet

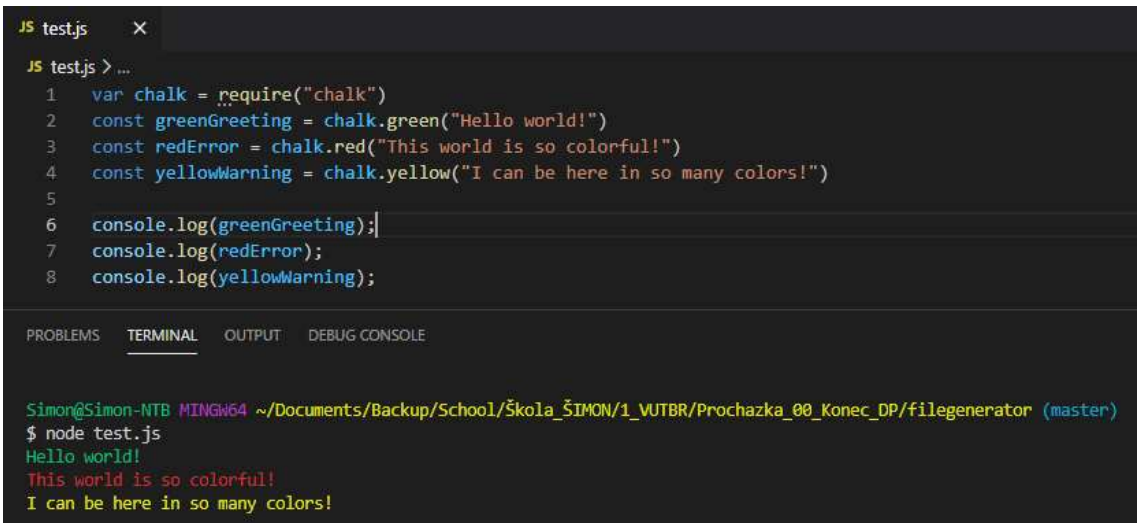
Modul figlet využívám pouze pro kosmetické úpravy, aby byla příjemnější práce s příkazovým řádkem. Tento modul umožňuje generovat bannery uvnitř příkazové řádky

na základě symbolů. Příkladem použití může být například informativní zpráva po nainstalování modulu či správném spuštění. (26)

1.4.8 chalk

Pro kosmetické úpravy využívám ještě další modul, který se nazývá *chalk*. Tento balíček umožňuje zabarvovat hlášky v konzoli a udělat tak práci pro uživatele příjemnější. Ukázku tohoto balíčku je možné vidět na obrázku 14, kde zbarvuji 3 řádky textu různými barvami. (27)

V aplikaci tyto kosmetické úpravy využívám k zobrazení chybových hlášek či informování uživatele. Například bezproblémové spuštění programu a jeho dokončení se vypíše zeleně, zatímco chyby se objeví v červené barvě. Zprávy s informativních charakterem, který by si mohl vyžadovat pozornost uživatele, využívají žlutou barvu.



Obrázek 14: Příklad využití knihovny chalk (Zdroj: vlastní zpracování)

1.4.9 mssql a msnodesqlv8

Diplomová práce vyžaduje práci s databází, neboť je nutné vygenerovat soubory na základě definicí systémových procedur. Pro tyto účely slouží dva balíčky a jejich kombinace umožňuje pracovat s Microsoft SQL Serverem a databázemi v něm.

Prvním z těchto balíčků je *mssql*. Tento balíček je klient pro připojení k Microsoft SQL Serveru a jedná se pouze o jednu z mnoha možností, jak připojení vytvořit. Mezi jiné možnosti patří například Sequelize či jiné nástroje pro objektově relační mapování. (28)

Druhým balíčkem je *msnodesqlv8*, což je řadič pro tabulkový datový proud. Vzhledem k tomu, že výsledky databází jsou zobrazené v tabulkách, je nutné použít některý z dostupných řadičů pro správné získání dat. (29)

1.5 Databáze

V rámci diplomové práce se databázemi zabývám pouze okrajově. Není třeba ukládat data, pouze získat popis databázových struktur. Z toho důvodu v této kapitole krátce popisuji databáze, konkrétně Microsoft SQL Server, neboť právě tento systém je využíván společností LOGEX.

1.5.1 Co je to databáze

Databáze je organizovaný soubor strukturovaných informací (dat), které jsou uloženy v elektronické podobě v počítači. Takový soubor je obvykle řízený systémem řízení báze dat. (30)

Data jsou nejčastěji ukládána do tabulek, kde každý řádek představuje jeden záznam, sloupce pak definují strukturu záznamu. Díky tomu je možné efektivní zpracování a vytváření dotazů, které zajišťují přístup a správu dat. Většina databází využívá k zadávání dat dotazovací jazyk SQL, existují však také NoSQL databáze. (30; 31)

1.5.2 SQL

Structured Query Language je dotazovacím jazykem, který používají téměř všechny relační databáze pro vytváření dotazů, manipulaci s daty, jejich definice a řízení přístupu. Jazyk byl prvně vyvinutý společností IBM ve spolupráci s Oracle a dal za vznik několika rozšířením, jako třeba Transact-SQL od Microsoftu, MySql od Oracle či PostgreSQL. (30)

1.5.3 Systém řízení báze dat

Databáze často využívá pro správu softwarový program, kterému se říká systém řízení báze dat (*DBMS*, z anglického *Database Management System*). Jedná se o rozhraní mezi databází a koncovým uživatelem, které umožňuje uživatelům získávat či upravovat

databázi a určovat, jakým způsobem jsou informace v ní organizovány. Příkladem takového systému může být Microsoft Access či Microsoft SQL Server. (30)

1.5.4 Microsoft SQL Server

Microsoft SQL Server je relační databázový systém vyvinutý společností Microsoft. Je postavený na jazyku SQL a využívá implementaci Transact-SQL, která rozšiřuje standardní jazyk o několik konstrukcí a umožňuje nové funkcionality. (32)

V návrhu řešení se konkrétně zabývám systémem Microsoft SQL Server v edici Express, který využívám pro definici nových tabulek a uložených procedur. (33)

1.6 Angular CLI

Vzhledem k tomu, že vyvíjím program, který doposud neexistuje, nelze využít žádné z analytických metod. V analytické části práce se zabývám podobnými nástroji, které jsem vyhledal za účelem inspirace pro tvorbu vlastního řešení. Jedním z těchto nástrojů je Angular CLI.

Angular je vývojová platforma napsaná v jazyce TypeScript, která obsahuje framework, kolekci propojených knihoven, a vývojářské nástroje které pomáhají sestavit, otestovat a upravovat zdrojový kód. Tento framework je využitý primárně pro tvorbu webových aplikací. Mezi jeho knihovnami však můžeme najít rozhraní v prostředí příkazového řádku, které usnadňuje vývoj nových aplikací. (34)

Tato příkazová řádka umožňuje snadné generování souborů v pracovním adresáři za využití příkazu *ng*. Obsahuje mnoho různých užitečných příkazů, mezi které patří například *new*, který vygeneruje v současném umístění nové složky a soubory, které definují základní strukturu projektu. (34)

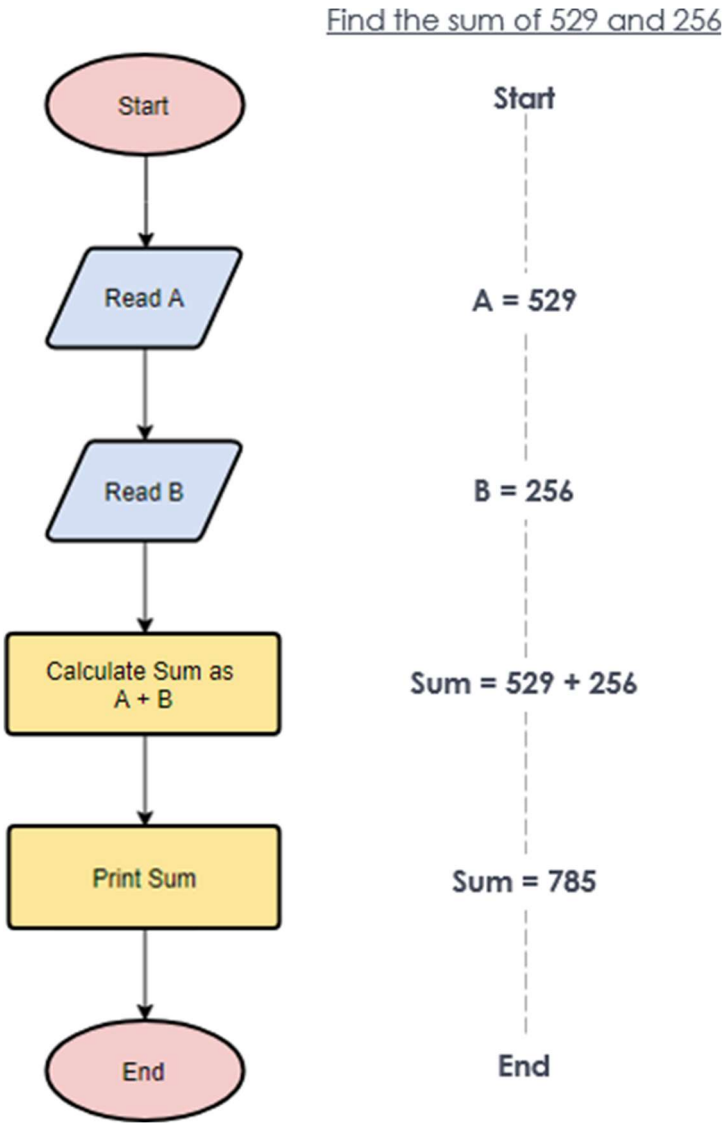
V diplomové práci využívám inspiraci zejména z této knihovny. Výsledný program je velmi podobný tomuto prostředí. Je možné mít jej nainstalovaný jak globálně, tedy volat jej z jakéhokoliv adresáře, nebo využívat jednu verzi pro každý projekt zvlášť.

1.7 Vývojový diagram

Posledním nástrojem, který v práci používám, je vývojový diagram. Tento nástroj slouží ke grafickému znázornění jednotlivých kroků algoritmu a je sestavený

z jednoduchých geometrických útvarů, které jsou propojené šipkami, jež definují posloupnost postupu. (35)

V tomto diagramu má každý geometrický útvar svoji funkci. Elipsa značí začátek a konec programu a obdélník s kulatými rohy definuje činnost. Kosodélník umožňuje definovat vstup a výstup. Pro rozhodovací proces je možné využít kosočtverec, díky kterému jsme schopni rozvětvit program v případě, že je nutné definovat podmínku. (35)



Obrázek 15: Příklad vývojového diagramu pro součet dvou čísel (Zdroj: 35)

Na obrázku 15 je možné vidět jednoduchý vývojový diagram pro součet dvou čísel, konkrétně 529 a 256. Začátek a konec programu je definovaný elipsou. Po začátku následuje čtení dvou čísel za využití kosodélníku. Proces je zakončen dvěma operacemi – součtem samotných čísel a jejich výpisem.

2 ANALÝZA PROBLÉMU A SOUČASNÉ SITUACE

Diplomová práce se zabývá vytvořením nástroje, který bude generovat zdrojový kód pro přístup k databázové vrstvě, tedy vytvoří soubory, které popisují databázové objekty. Tento nástroj tvořím pod záštitou firmy LOGEX. V analytické části z toho důvodu nejprve představuji společnost a poté se představím důvody, proč je projekt nutné vytvořit. Kromě společnosti a analýzy existujícího nástroje společnosti se také zabývám dalším nástrojem zvaným *Angular CLI*, kterým jsem se při tvorbě nástroje inspiroval.

2.1 Analýza společnosti LOGEX Solution Center s.r.o.

Společnost LOGEX Solution Center byla založena 19. května 2017 a jedná se o dceřinou společnost holandské společnosti LOGEX Group B.V. Níže je možné nalézt údaje o firmě z obchodního rejstříku. (36)

Název: LOGEX Solution Center s.r.o.

Sídlo: Nové sady 996/25, Staré Brno, 602 00 Brno

Právní forma: Společnost s r.o.

Datum založení: 19. květen 2017

IČO: 06111904

Předmět podnikání: výroba, obchod a služby neuvedené v přílohách 1 až 3 živnostenského zákona

Společnost je vývojovým střediskem pro holandskou firmu, která se zabývá analytikou ve zdravotnictví. Samotná firma není zisková, neboť veškeré příjmy plynou z činnosti mateřské společnosti.

LOGEX působí na trhu již od roku 2008 a vyvíjí analytický software pro holandské nemocnice. V roce 2017 vytvořila vývojové středisko v Brně, kde nyní probíhá vývoj většiny produktů. V posledních letech se však několikanásobně zvětšila. Zatímco v roce 2018 měla společnost jako celek zhruba 50 zaměstnanců, v současnosti jich je více než 300 a stále se rozšiřuje. Společnost také spojila síly s jinými firmami jako Ivbar ve Švédsku či Prodacapo ve Finsku. Důvodem k tomuto spojení je mimo jin snaha o průnik do mezinárodního prostředí. (37)

2.2 Analýza nástroje Data Access Generator

V rámci diplomové práce se zabývám přístupu k datové vrstvě softwarové aplikace. Vzhledem k tomu, že veškeré produkty firma vyvíjí interně a je jich velké množství, pro usnadnění práce si svépomocí vytvořila nástroje. Jedním z těchto nástrojů je webová aplikace *Data Access Generator*, který se využívá ke generování zdrojového kódu pro přístup k databázi. Tato část více rozebírá tento firemní nástroj.

2.2.1 Popis nástroje

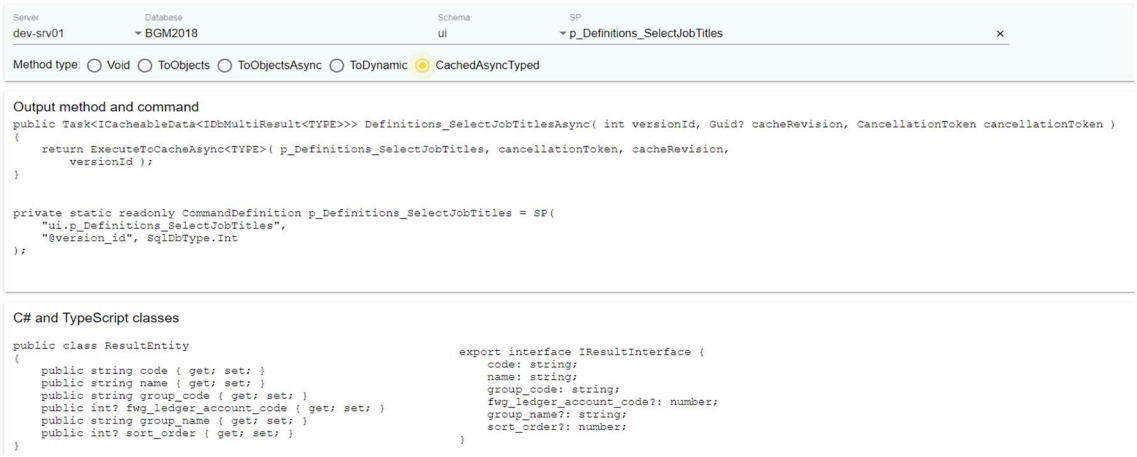
Data Access Generator společnosti se využívá ke generování zdrojového kódu pro přístup k datům. Jedná se o poměrně jednoduchý nástroj, který umožňuje 5 vstupů, podle kterých vygeneruje zdrojový kód pro C# a TypeScript soubory. Pro vytvoření zdrojového kódu jsou nutné již zmíněné vstupy. Tyto vstupy jsou:

- Server – vzdálený počítač, na kterém je databáze uložena
- Database – Jméno databáze, ze které má vygenerovat data
- Schema – Databázové schéma, ve kterém se nachází uložená procedura
- SP – Název uložené procedury, pro kterou se má zdrojový kód vygenerovat
- Method type – Metoda, kterou má vygenerovaný kód používat. Jedná se o interní metody společnosti.

Přestože je tento nástroj velmi užitečný, je zapotřebí znalý uživatel, který ví, jak se s nástrojem pracuje. Pro práci je také potřeba velké množství manuální práce – nástroj negeneruje soubory přímo v projektu, ale vygeneruje textovou předlohu v prohlížeči. Uživatel tak musí s nástrojem neustále pracovat.

2.2.2 Prvky nástroje

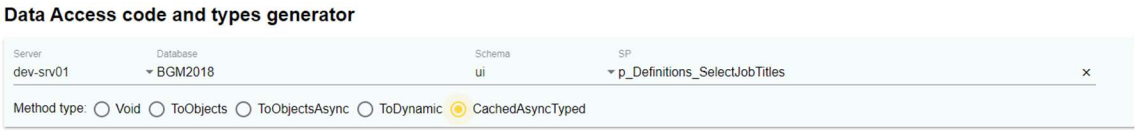
Nástroj se skládá ze dvou částí. První částí je v hlavičce nástroje výběr vstupů, tělo pak obsahuje zdrojový kód. Na obrázku 16 je možné vidět pohled na nástroj.



Obrázek 16: Data Access Generator (Zdroj: vlastní zpracování)

2.2.3 Hlavička nástroje

Jak již bylo zmíněno, hlavička nástroje obsahuje vstupy nutné ke generování zdrojového kódu. Tuto hlavičku je možné vidět na následujícím obrázku 17.



Obrázek 17: Hlavička nástroje Data Access Generator (Zdroj: vlastní zpracování)

2.2.3.1 Databázové parametry

První čtyři vstupy využíváme pro popis procedury. Díky parametrům *server*, *database*, *schema* a *SP* jsme schopni zavolat takovou proceduru v případě, že nemá parametry.

```
EXEC [SERVER].[DATABASE].[SCHEMA].[STORED_PROCEDURE]
```

Pokud by uložená procedura měla parametry, tento kód by neprošel, protože jí parametry musíme předat. *Data Access Generator* však proceduru nevolá, pouze získá její popis. Toho jsme schopni dosáhnout za využití systémové procedury *[sys].[p_describe_first_result_set]*.

Systémová procedura *[sys].[p_describe_first_result_set]* totiž umožňuje získat popis všech výstupních sloupců procedury. Za využití této procedury je tak možné generovat typové soubory výstupu. Příkladem volání procedury je následující kód.

```
EXEC sys.sp_describe_first_result_set N'SERVER.DATABASE.SCHEMA.STORED_PROCEDURE'
```

Výše uvedený kód tedy zavolá systémovou proceduru, která popisuje výsledné hodnoty jiné procedur. Jméno procedury, kterou chceme popsat, je parametrem systémové procedury.

2.2.3.2 Generátorové parametry

Pátý a poslední parametr, který nástroj využívá, je metoda. Metoda v tomto kontextu znamená způsob, jakým chceme zacházet s výstupem procedury. Některé metody jsou specifické pro společnost LOGEX, neboť její backendový framework umožňuje různé způsoby zpracování procedur. Například metoda *CachedAsyncTyped* zavolá proceduru takovým způsobem, aby se uložila do cache, zatímco *ToObjects* pouze proceduru zavolá a vrátí její výsledky bez dalších zásahů. Metoda tak žádným způsobem neovlivní volání databázové procedury, definuje ale následné akce, které se mají stát uvnitř C# vrstvy. Podporované metody jsou:

- *Void* – Definuje generátor bez výsledných hodnot.
- *ToObjects* – Definuje generátor s typovanými výslednými hodnotami, předpokládá synchronní volání.
- *ToObjectsAsync* – Definuje generátor s typovanými výslednými hodnotami, předpokládá asynchronní volání
- *ToDynamic* – Definuje generátor bez typovaných výsledných hodnot, předpokládá synchronní volání.
- *CachedAsyncTyped* – Definuje generátor s typovanými výslednými hodnotami, předpokládá asynchronní volání a uložení výsledku do cache.

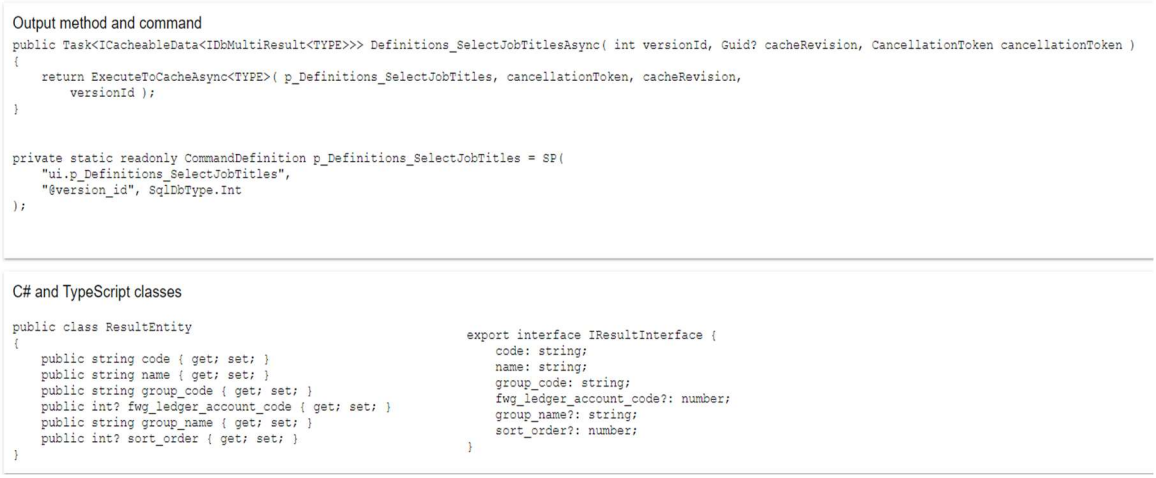
Generátorové parametry jsou jednou ze slabých stránek nástroje. V průběhu let se totiž rozšířily metody volání, nicméně generátor nebyl upraven tak, aby tyto metody podporoval.

2.2.4 Tělo nástroje

Druhá část nástroje je samotný zdrojový kód. Zdrojový kód je vygenerovaný na základě všech vstupů a jeho sestavení se děje na základě systémových procedur SQL serveru.

Tělo se skládá ze dvou částí. První částí je zdrojový kód definice a její volání, druhá pak obsahuje zdrojový kód pro typy v jazyce C# a TypeScript. Výstup procedury

p_Definitions_SelectJobTitleAsync se vstupním parametrem *versionId* a metodou *CachedTypedAsync* je možné vidět na obrázku 18.

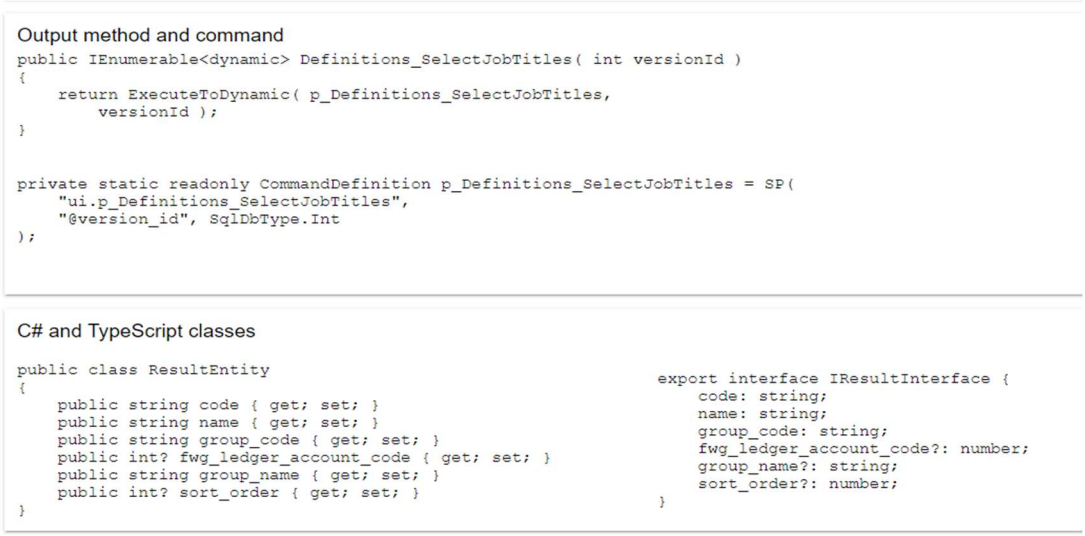


Obrázek 18: Výsledek nástroje Data Access Generator cache metodou (Zdroj: vlastní zpracování)

2.2.4.1 Definice volané procedury

První část výstupu obsahuje 2 členy. Prvním členem je metoda *Definitions_SelectJobTitlesAsync*, která pouze zavolá proceduru. Druhý člen je pak samotná definice procedury, která obsahuje jméno a všechny její parametry.

Tato část se mění na základě metody. V případě *CachedAsyncTyped* se tedy ukládá výsledek do cache, což zajišťuje interní funkce *ExecuteToCacheAsync*. Obrázek 19 obsahuje stejné volání, pouze s generátorovou metodou *ToDynamic*. Jak lze vidět, jediný rozdíl ve volání je ve volané funkci *ExecuteToDynamic*.



Obrázek 19: Výstup nástroje Data Access Generator dynamic metodou (Zdroj: vlastní zpracování)

2.2.4.2 Zdrojový kód typů

Zdrojový kód pro typové soubory obsahuje definici třídy pro C#, v TypeScriptu je pak využitý *interface*. Vzhledem k tomu, že každý databázový typ je možné nějakým způsobem přeložit na C# či TypeScriptový typ, vytvoření takového kódu je velmi jednoduché. Například databázový typ *varchar* je možné přeložit na C# či TypeScriptový *string*. Mírně složitější převody jsou číselných typů, pro něž lze využít jednoduchý slovník.

Typové soubory mají stejnou strukturu bez ohledu na to, která metoda je vybraná. Z toho důvodu i metody které nevrací žádný výsledek mohou mít definovaný výsledek v nástroji. Jedná se o další slabou stránku nástroje – tedy bezvýsledné metody by neměly mít definovaný výsledek a nemělo by být možné je volat jiným způsobem než *Void*. Opačné pravidlo platí pro metody, které výsledek vracejí – takové metody by nemělo být možné volat metodou *Void*.

2.2.5 Nedostatky nástroje

V průběhu analýzy jsem již zmínil dva nedostatky, které se snažím v této diplomové práci vyřešit. Prvním nedostatkem jsou generátorové parametry, které se od poslední úpravy rozšířily. Tyto parametry jsou definované společností a jedná se o firemní požadavek. Druhým nedostatkem je matoucí možnost volání metod, tedy že procedury, které neobsahují výstup, je možné volat metodou, která jej vyžaduje, a naopak.

Dalším nedostatkem nástroje je jeho webové rozhraní. Přestože nástroj správně generuje a kód je možné ihned využít, vývojář, který chce tento nástroj používat, musí nejprve otevřít stránku, definovat všechny parametry, a výsledek si pak musí překopírovat manuálně do svého počítače. Přestože někteří vývojáři mohou tento způsob upřednostňovat, většina by raději využila příkazovou řádku, která umí generovat soubory. Z toho důvodu je vytvoření příkazové řádky hlavním cílem této diplomové práce.

2.3 Jiné nástroje generující zdrojový kód

V poslední kapitole analytické části se zabývám dalšími existujícími generátory zdrojového kódu, zejména takové, které vytvoří složkovou strukturu i soubory. Z mnoha různých generátorů jsem vybral konkrétně Angular CLI.

2.3.1 Nástroje generující zdrojový kód přístupu k databázi

Z hlediska nástrojů generujících zdrojový kód pro vrstvu *Data Access* se mi nepodařilo dohledat dostatečně kvalitní nástroj. Přestože některé nástroje existují – například Entity Framework tuto funkcionalitu umožňuje – žádný z nich nepodporuje vlastní úpravy. Z toho důvodu mimo jiné firma vytvořila vlastní nástroj pro takové generování.

2.3.2 Angular CLI

Angular CLI je rozhraní příkazové řádky pro framework Angular, který slouží k vývoji webových aplikací. Samotná příkazová řádka je pouze pomocný nástroj pro vývojáře, který umožňuje generování základních částí programu (tzv. *scaffolding*). Tento nástroj jsem si vybral z toho důvodu, že firma rozsáhle používá tento framework, a z toho důvodu se jeví jako vhodná inspirace, protože můžu příkazovou řádku přizpůsobit tak, aby byla povědomá. Angular CLI může být nainstalované buď globální (v systému počítače), lokálně (v projektu) či kombinovaně. V případě kombinace je využité primárně lokální verze.

Příkazová řádka obsahuje velké množství metod pro generování různých souborů. Například příkaz *ng generate component* vygeneruje pouze jednu část projektu, zatímco *ng new* vygeneruje celý projekt. V rámci analýzy se nezabývám funkcionalitou této příkazové řádky, ale spíše možnosti konfigurace. Příkazová řádka totiž musí umožňovat všechny vstupy, které obsahuje současné webové rozhraní, dále však musí mít možnost generování souborů do konkrétní složky, respektive stanovení cesty k výsledným souborům.

2.3.2.1 Parametrizace příkazu *ng generate component*

Jak jsem již zmínil, *ng generate component* umožňuje vygenerovat soubor, případně více souborů, na základě konfigurace uvnitř příkazové řádky.

Parametry můžeme přidat buď skrze celá jména, nebo jako *alias* pro kratší zápis. Na základě těchto parametrů pak Angular vygeneruje pro konkrétní konfiguraci více souborů, jiné pak definují vnitřní strukturu souborů.

Pro generování více či méně souborů můžeme například použít možnost *inlineStyle*, která definuje, zda se mají vytvořit soubory pro kaskádové styly či nikoliv. Na následujícím kódu je možné vidět 4 způsoby generování. První dva řádky definují volání v plné podobě, zatímco druhý zápis je identický prvním dvěma řádkům z pohledu funkčnosti, využívá však *alias*.

```
ng generate component componentName --inlineStyle=true
ng generate component componentName --inlineStyle=false
ng g c componentName -s=true
ng g c componentName -s=false
```

2.3.2.2 Globální parametrizace

Globální parametrizace příkazové řádky *Angular CLI* je prováděna pomocí příkazu *ng config*. Příkaz *ng config* bere parametr *global*, který definuje, zda chceme zobrazit či pozměnit globální konfiguraci či nikoliv.

Konfigurace je uložená v souboru typu *JSON* a díky tomu existují 2 možnosti konfigurace – skrze příkazovou řádku, anebo úpravou *JSON* souboru.

2.4 Celkové shrnutí analýzy

V analytické části diplomové práce jsem se zabýval společností LOGEX, pro kterou je diplomová práce zpracována, a dalšími dvěma nástroji. Prvním z těchto nástrojů byl generátor zdrojového kódu společnosti LOGEX.

Ze silných stránek lze vybrat zejména fakt, že nástroj vygeneruje plně funkční kód, který lze zkopírovat a přímo spustit, avšak vyžaduje aktivní zapojení vývojáře, zejména při vytváření třídy *Controller* pro zachycení dotazu.

Významnou slabinou je možnost přístupu pouze skrz webové prostředí. Vývojář musí pro práci s nástrojem otevřít konkrétní webovou adresu, specifikovat vstupy a poté zkopírovat výsledky k sobě do počítače. Dále existují matoucí možnosti volání metod – například vygenerování výstupních typů pro funkci, která žádné nemá.

Druhým analyzovaným nástrojem byla příkazová řádka *Angular CLI*. Tímto prostředím se v diplomové práci velmi inspiruji, neboť firma LOGEX tuto příkazovou řádku velmi využívá, a je vhodné, aby vytvoření nástroj byl co nejvíce povědomý.

Z hlediska analýzy jsem se nezabýval funkčností programu, neboť generované soubory jsou velmi odlišné od těch, které by měly být vygenerované mým nástrojem. Zvláštní pozornost jsem věnoval parametrizací příkazové řádky, tedy jakým způsobem je možné specifikovat dodatečné požadavky programu. Z tohoto nástroje ve své diplomové práci aplikuji zejména možnosti předávání parametrů příkazové řádce, využití *aliasů* a uložení konfigurace do souboru *JSON*.

3 VLASTNÍ NÁVRH ŘEŠENÍ, PŘÍNOS PRÁCE

Na základě výsledků analýzy současného stavu vyplývají 3 nedostatky nástroje společnosti LOGEX. Tyto nedostatky dále vstupují jako požadavky do vlastního návrhu řešení. V rámci řešení se tedy budu zabývat následujícími úpravami současného nástroje:

- Vylepšení současných generátorových parametrů
- Změna výsledku volaných metod pro lepší srozumitelnost
- Vytváření souborů v počítači

V této práci se zabývám novou aplikací v rozhraní příkazové řádky, protože z webového rozhraní není možné přistupovat do souborového systému počítače. Řeším tak poslední bod z výše zmíněných tří úprav. Ostatní dva požadavky pouze beru v potaz při vývoji této aplikace a usiluji o to, aby byly podporované všechny generátorové parametry a výsledky byly jasné.

3.1 Databáze pro práci s nástrojem

Abych byl schopný pracovat s nástrojem, je nutné si vytvořit vlastní databázi. Společnost LOGEX pracuje s Microsoft SQL Serverem, je tedy vhodné, aby moje diplomová práce využívala stejné nástroje. K vytvoření databázového systému jsem využil Microsoft SQL Server 2019 Express na svém počítači na adrese *localhost*. Pro jeho úpravy využívám Microsoft SQL Server Management Studio 18.

V rámci diplomové práce není nutné navrhnout plně funkční databázi. Tabulky nemusí být propojené a není potřeba žádná dodatečná funkcionalita. Pro fungování generátoru je nutné, aby existovaly databáze, schéma a v nich uložená procedura. Z toho důvodu se v této části zabývám pouze vytvořením dvou databází, kde každá databáze obsahuje schéma a dvě tabulky, tedy celkem 4 tabulky.

Z hlediska uložených procedur jsem vytvořil pro testovací účely pouze 8 procedur. Polovina těchto procedur slouží pouze k získání všech dat, které tabulky obsahují. Druhá polovina pak umožňuje parametrizované přidání záznamů do tabulek. Tím pokrývám různé možnosti využití generátoru.

3.1.1 Vytvoření databází a tabulek

Vytvoření databáze je možné dosáhnout příkazem *CREATE DATABASE*. Při další práci s touto databází je nutné ji používat skrze příkaz *USE*. Při vytváření tabulek jsem vytvořil pro obě databáze identické tabulky. Z pohledu tabulek není nutná zvláštní funkcionalita. Při jejich vytváření jsem však definoval 5 sloupců, a to každý s jiným typem. Je to z toho důvodu, že C# obsahuje jiné typy než databáze, a to platí i pro TypeScript. Tento rozdíl je nutné brát v potaz a v samotné aplikaci musíme tyto typy měnit. Tabulky jsem rozdělil v každé databázi do dvou schémat – výchozí schéma *dbo* a mnou vytvořená *dev*. Kód pro vytvoření databází a tabulek je možné najít v příloze 1 a výsledkem je vytvoření následujících tabulek.

- *[DataAccessGeneratorDB].[dbo].[Test_Table]*
- *[DataAccessGeneratorDB].[dev].[Test_Table_Second]*
- *[DataAccessGenerator_v2].[dbo].[Test_Table]*
- *[DataAccessGenerator_v2].[dev].[Test_Table_Second]*

Každá tabulka obsahuje jediný záznam pro vyzkoušení funkčnosti tabulek. Pro vložení těchto prvních záznamů jsem využil kód, který je možné vidět v příloze 2.

3.1.2 Vytvoření uložených procedur

Pro vytvoření uložených procedur jsem použil příkaz *CREATE PROCEDURE*. Každou proceduru jsem uložil do schématu *ui*, které definuje, že se schéma volá z uživatelského rozhraní (user interface).

První čtyři procedury, které jsem vytvořil, slouží pouze k vrácení všech údajů z prvních čtyř tabulek. Pro tuhle funkcionalitu jsem pouze přidal příkaz *SELECT * FROM*, který vybere všechny záznamy v tabulce a vrátí je. Kód pro tyto procedury je možné najít v příloze 3 a jejich jména jsou následující:

- *[DataAccessGeneratorDB].[ui].[fetch_main_data]*
- *[DataAccessGeneratorDB].[ui].[fetch_main_data_second]*
- *[DataAccessGenerator_v2].[ui].[fetch_main_data]*
- *[DataAccessGenerator_v2].[ui].[fetch_main_data_second]*

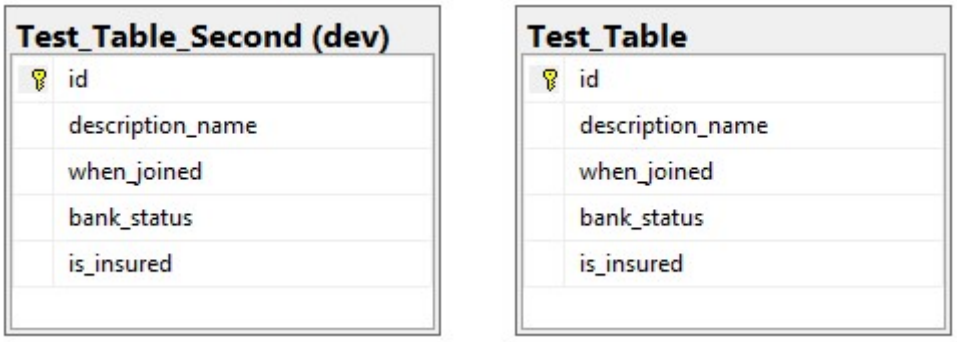
Druhá polovina uložených procedur umožňuje vložení hodnot do výše zmíněných tabulek. Celkem umožňuje pět parametrů, a to *id*, *description_name*, *when_joined*, *bank_status* a *is_insured*. Tyto parametry mají každý různé datové typy. Výsledkem jsou procedury vytvořené v příloze 4:

- *[DataAccessGeneratorDB].[ui].[add_main_data]*
- *[DataAccessGeneratorDB].[ui].[add_main_data_second]*
- *[DataAccessGenerator_v2].[ui].[add_main_data]*
- *[DataAccessGenerator_v2].[ui].[add_main_data_second]*

3.1.3 Shrnutí databázové části

V této podkapitole jsem se zabýval vytvořením databází, tabulek, schémat a uložených procedur pro práci s *DataAccessGenerator* nástrojem. V rámci diplomové práce není nutné, aby tyto tabulky byly propojené a aby uložené procedury měly nějakou složitější funkcionalitu.

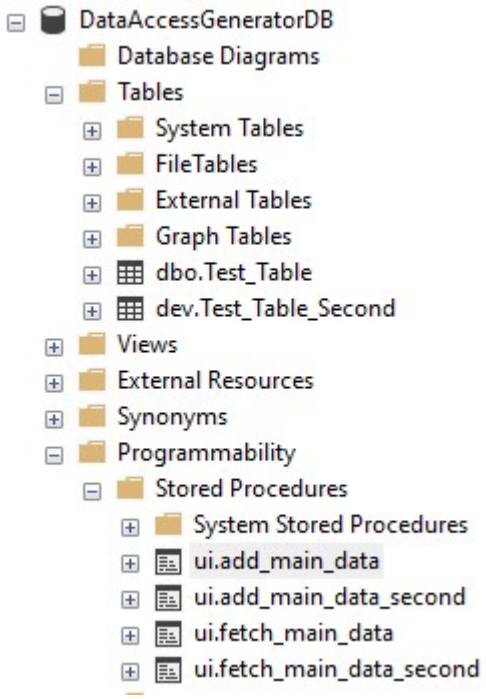
Diagram databáze DataAccessGeneratorDB je možné vidět na obrázku 20. Jelikož databáze tabulky v databázi nemusí mít žádné vztahy, je nutné pouze jejich existence a základní procedury, na diagramu nenalezneme žádné propojení mezi tabulkami.



Obrázek 20: Diagram databáze DataAccessGeneratorDB (Zdroj: vlastní zpracování)

Výsledkem je jeden databázový systém, který obsahuje dvě databáze, z nichž každá obsahuje dvě tabulky, celkem tedy 4 tabulky ve schématech *dbo* a *dev*. Dále jsem vytvořil 8 uložených procedur, z nichž polovina slouží pro získání dat z vytvořených tabulek, druhá polovina pak umožňuje přidávání záznamů do nich.

Pro vyzkoušení funkčnosti procedur jsem nejprve zavolal procedury pro vkládání záznamů, a poté jsem zavolal procedury, které získávají všechny záznamy z tabulek. Výsledek prošel testem a v příloze 5 je možné vidět kód, který vkládá druhý záznam do každé z tabulek, a poté pro každou tabulku vrátí 2 záznamy. Pro tuto funkcionalitu jsou využity právě uložené procedury. Všechny vytvořené struktury je možné vidět na obrázku 21, který obsahuje celou strukturu databáze.



Obrázek 21: Struktura databáze DataAccessGeneratorDB (Zdroj: vlastní zpracování)

3.2 Přístup k databázi v příkazové řádce

V následující kapitole se zabývám samotným vytvořením příkazové řádky. V první části zmiňuji všechny nástroje, které jsem využil pro vytvoření příkazové řádky. V dalších podkapitolách se zabývám samotným vývojem programu.

3.2.1 Využité nástroje

Při vytváření příkazové řádky jsem využil Node.js pro JavaScriptový runtime a programovací jazyk TypeScript pro vytvoření zdrojového kódu. Také jsem využil několik modulů třetí strany.

Vzhledem k tomu, že Node.js je určený pro JavaScript, pro kompilaci TypeScriptu jsem využil knihovnu *ts-node* a *typescript*. Připojení k databázi řeší moduly *mssql* a

msnodesqlv8, které umožňují z Node.js volat databázové příkazy. Parametrizace příkazové řádky je řešena využitím knihovny *yargs* a pro vypisování výsledků v příkazové řádce jsem také použil modul *chalk*, který umožňuje kosmetické úpravy. Vytváření souborů nakonec realizuje knihovna *fs-extra*.

3.2.2 Vývojový diagram

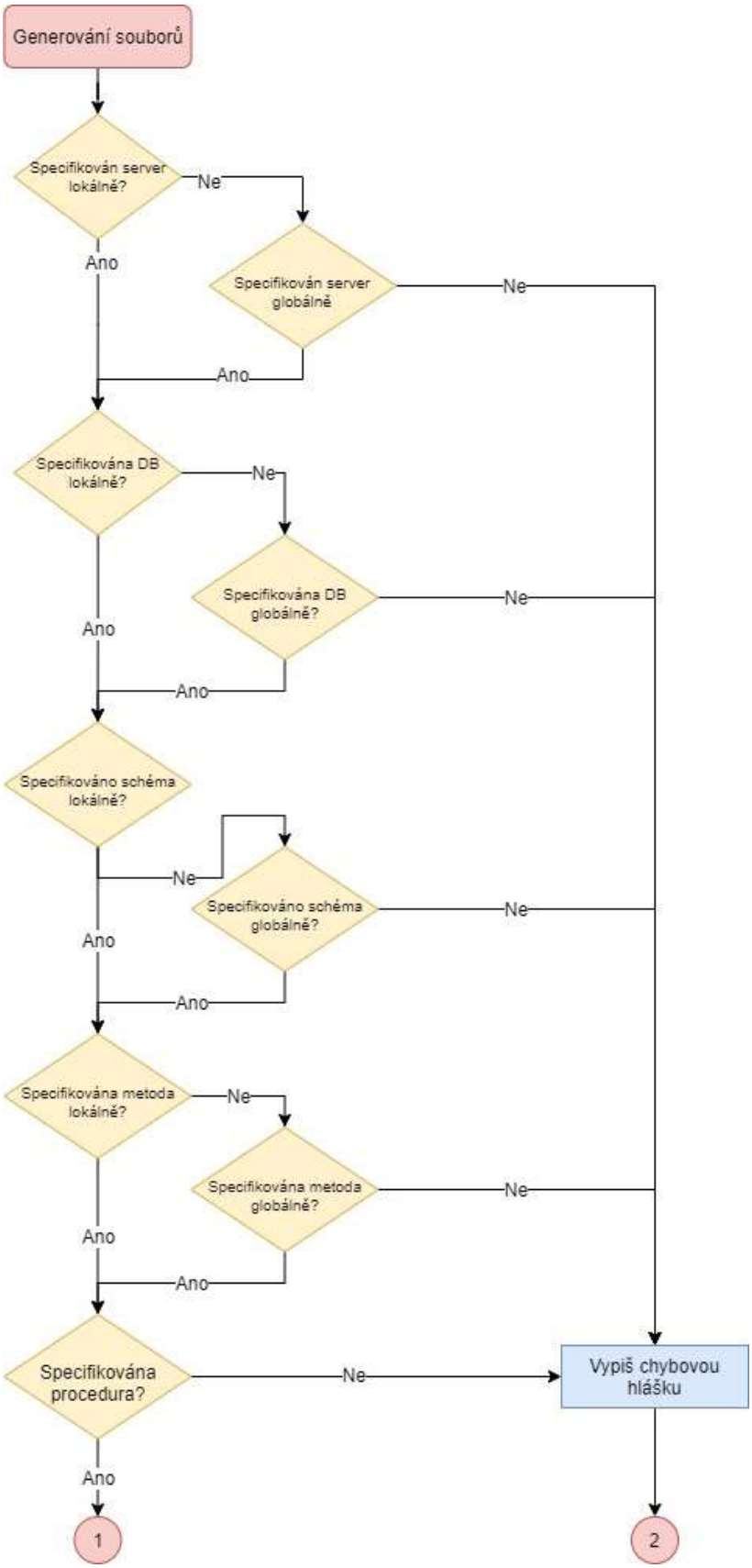
V této podkapitole se zabývám vývojovým diagramem příkazové řádky. Vzhledem k tomu, že vývojový diagram celého programu by byl příliš velký, rozhodl jsem se jej rozdělit na dvě části – zpracování povinných parametrů a zpracování parametrů dodatečných. Vývojový diagram bychom dále mohli rozšířit o vytvoření jednotlivých souborů, avšak tímto procesem se zabývám v jednotlivých kapitolách.

3.2.2.1 Vývojový diagram zpracování povinných parametrů

Prvním vývojovým diagramem je zpracování povinných parametrů. Každý generovaný soubor má základních 5 parametrů, které musí obsahovat každé volání. Tyto volání jsem již zmínil v analytické části. Pro jednoduchost je zde zopakuji.

- Server – vzdálený počítač, na kterém je databáze uložena
- Database – Jméno databáze, ze které má vygenerovat data
- Schema – Databázové schéma, ve kterém se nachází uložená procedura
- SP – Název uložené procedury, pro kterou se má zdrojový kód vygenerovat
- Method type – Metoda, kterou má vygenerovaný kód používat. Jedná se o interní metody společnosti.

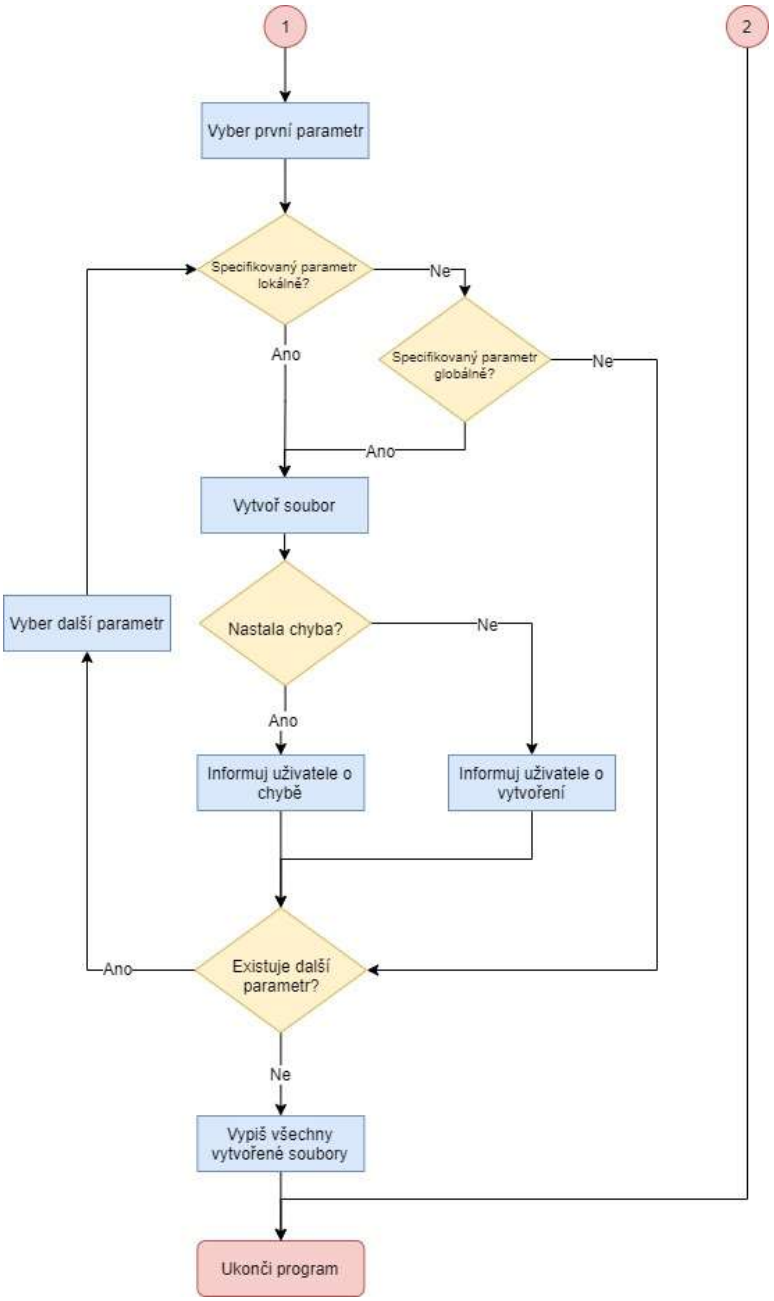
V případě, že některý z parametrů chybí jak v lokální, tak globální konfiguraci, vypíše se chybová hláška. V opačném případě program bude pokračovat v generování. Vývojový diagram je možné vidět na obrázku 22.



Obrázek 22: Vývojový diagram pro zpracování povinných parametrů (Zdroj: vlastní zpracování)

3.2.2.2 Vývojový diagram pro zpracování dodatečných parametrů

Druhým vývojovým diagramem je zpracování dodatečných parametrů. Rozdíl v jejich zpracování je v tom, že v případě jejich absence se program neukončí. V případě, že existují, tak na jejich základě vytvoří soubor. Až poté, co se zkontrolují všechny parametry, program se ukončí. Z toho důvodu se jedná o jakousi smyčku. Vývojový diagram je možné vidět na obrázku 23.



Obrázek 23: Vývojový diagram pro zpracování dodatečných parametrů (Zdroj: vlastní zpracování)

3.2.3 Přístup k databázovému systému

Prvním bodem, kterým se zabývám, je přístup k databázovému systému. Jak jsem již zmínil v podkapitole 3.2.1, pro tuto funkcionalitu využívám knihovny *mssql* a *msnodesqlv8*. Tyto knihovny umožňují připojení k databázi skrze tzv. *ConnectionPool*.

```
const pool = new sql.ConnectionPool(config);
```

Výše uvedený zdrojový kód vytvoří připojení k Microsoft SQL Serveru na základě předem definované konfigurace, která obsahuje název serveru a databáze, které jsou potřeba k připojení.

Vzhledem k tomu, že každý příkaz využívá databázové volání, neboť se jedná o generátor databázových metod, je nutné tuto konfiguraci mít vždy definovanou. Z toho důvodu je vhodné, aby vždy byla definována nějaká defaultní konstanta, kterou je možné za běhu měnit.

Rozhodl jsem se tedy vytvořit konfigurační soubor, který bude možné měnit a tyto změny budou trvale uložené. Z tohoto souboru se bude brát defaultní hodnota pro databázový přístup. Pokud bude databáze poslána jako parametr generování, bude využita tato hodnota. Pokud nebude definovaná defaultní hodnota ani parametr, program vypíše chybovou hlášku. Zdrojový kód pro připojení k databázi je možné najít v příloze 6.

3.2.4 Získání dostupných databází

Po připojení programu k databázi je možné využívat databázových příkazů. První z těchto příkazů je dotaz na databáze. Tyto databáze je možné najít v systémové tabulce zvané *databases*, a tak se jedná o přímočarý dotaz. Seznam všech dotazů v zápisu TypeScriptových funkcí, které jsou v diplomové práci využity, je možné nalézt v příloze 7.

Pro získání dostupných databází je tedy využitý jednoduchý dotaz na tabulku. Vzhledem k tomu, že je nutné znát pouze jména databází, výsledek jsem omezil na sloupec *name* a je možné jej vidět na obrázku 24.

	name
1	master
2	tempdb
3	model
4	msdb
5	DataAccessGeneratorDB
6	DataAccessGenerator_v2

Obrázek 24: Výsledek dotazu na databáze (Zdroj: vlatní zpracování)

Jak je možné vidět, při vytvoření dvou vlastních databází vrací dotaz databází 6. Je to z toho důvodu, že první 4 z těchto databází jsou vytvořené automaticky, a pouze databáze od indexu 5 jsou vytvořené uživatelem. Pro eliminaci prvních čtyř databází jsem tedy nakonec obohatil příkaz o klauzuli *where*. Celý dotaz je možné nalézt v příloze 7, implementaci v příloze 8 a jeho výsledek je možné vidět v obrázku 25.

```
$ ts-node index.ts listDb
Resulting properties:
[
  { name: 'DataAccessGeneratorDB' },
  { name: 'DataAccessGenerator_v2' }
]
```

Obrázek 25: Výsledek získání databází (Zdroj: vlastní zpracování)

3.2.5 Získání dostupných schémat

Pro získání dostupných schémat je důležité znát databázi. Každá databáze má totiž informační schéma, které obsahuje všechny schémata, která jsou v databázi vytvořené.

V tomto případě můžeme opět využít jednoduchý dotaz, který vybere záznamy z tabulky *INFORMATION_SCHEMA.SCHEMATA*. I zde je však nutné výsledky vyfiltrovat, neboť tato tabulka obsahuje i sama sebe, respektive schema *INFORMATION_SCHEMA*, dotaz totiž vrací 13 předdefinovaných schémat. Z těchto schémat je však standardně v databázi standardně využité pouze jedno, a to schema *dbo*. V dotazu jsem opět využil pouze jméno schématu a je možné jej vidět na obrázku 26.

	SCHEMA_NAME
1	dbo
2	guest
3	INFORMATION_SCHEMA
4	sys
5	dev
6	ui
7	db_owner
8	db_accessadmin
9	db_securityadmin
10	db_ddladmin
11	db_backupoperator
12	db_datareader
13	db_datawriter
14	db_denydatareader
15	db_denydatawriter

Obrázek 26: Výsledek dotazu na schémata (Zdroj: vlastní zpracování)

Pro vyfiltrování dat je tedy nutné zbavit se všech databází, které začínají na *db_* a které se jmenují *guest*, *INFORMATION_SCHEMA* anebo *sys*. Finální verzi dotazu je možné najít v příloze 7, jeho volání je možné najít v příloze 9 a vrácené hodnoty na obrázku 27.

```
$ ts-node index.ts listSchema
Resulting properties:
[
  { SCHEMA_NAME: 'dbo' },
  { SCHEMA_NAME: 'dev' },
  { SCHEMA_NAME: 'ui' }
]
```

Obrázek 27: Výsledek získání schémat (Zdroj: vlastní zpracování)

3.2.6 Získání dostupných procedur

Procedury jsou uložené v databázi a je možné se k nim dostat na základě schématu. V tomto případě je opět nutné mít specifikovanou databázi, bez vybraného schématu však dostaneme větší množství procedur.

Pro získání procedur opět využijeme schéma *INFORMATION_SCHEMA*, avšak v tomto případě dotazujeme tabulku *ROUTINES*. Každá rutina má svůj typ, a v případě procedur můžeme tento typ omezit na *PROCEDURE*. Pro vybrání všech vytvořených procedur lze dotazovat výše zmíněnou tabulku s klauzulí *where*.

Vzhledem k tomu, že v této tabulce je také definované schéma, ve které se procedura nachází, je rozumné udělat další podmínku. V případě, že je schéma definované, výsledkem budou pouze procedury uložené v databázi pod konkrétním

schématem. Pokud schéma definované nebude, dotaz vrátí všechny uživatelem vytvořené procedury bez ohledu na schéma. V případě, že databáze nebude definovaná, program vyhodí chybu. Dotaz je možné najít v příloze 7 a pro získání těchto procedur volám funkci v příloze 10. Výsledek v podobě volání v příkazové řádce je možné vidět na obrázku 28.

```
$ ts-node index.ts listSp
Resulting properties:
[
  { ROUTINE_NAME: 'fetch_main_data_second' },
  { ROUTINE_NAME: 'fetch_main_data' },
  { ROUTINE_NAME: 'add_main_data_second' },
  { ROUTINE_NAME: 'add_main_data' }
]
```

Obrázek 28: Výsledek získání uložených procedur (Zdroj: vlastní zpracování)

3.2.7 Získání parametrů procedur

Předposlední částí databázové komunikace je získání parametrů procedur. Pro vytvoření aplikace je nutné znát veškeré parametry procedur, aby bylo možné ji zavolat. Pro popis těchto parametrů je možné se opět dotazovat na systémovou tabulku databáze *parameters*, která je součástí *sys* schématu. Tato tabulka obsahuje veškeré parametry všech procedur, které jsou v databázi uložené, a je nutné výsledek dotazu zredukovat pomocí klauzule *where*, kterým vyfiltrujeme všechny parametry, které nejsou spojené s procedurou.

Vzhledem k tomu, že vygenerované soubory mají být v TypeScriptu a C#, je nutné znát také datové typy. Tabulka *parameters* tyto typy obsahuje také, avšak neobsahuje jejich jména, pouze ID. Z toho důvodu je zapotřebí využít ještě další dotaz na tabulku *sys.types*, která vrátí všechny typy, které databáze využívá.

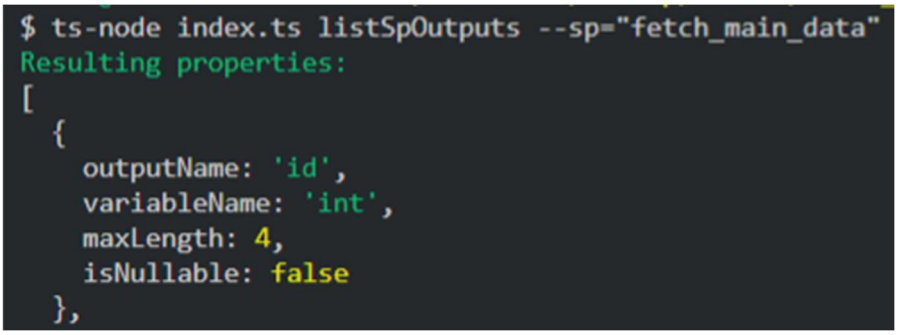
Zpracování výsledku probíhá již v aplikaci, kde mapujeme parametry s jejich typy. Dotazy je možné vidět v příloze 7 a jejich zpracování pak v příloze 11. Na obrázku 29 je pak možné vidět jeden parametr v konzoli

Obrázek 29: Výsledek získání parametrů procedury (Zdroj: vlastní zpracování)

3.2.8 Získání výstupů procedur

Poslední databázovou informací, kterou pro zpracování práce potřebuji, je popis výstupních sloupců procedury. Pro tento popis existuje také systémová možnost, avšak tentokrát se nejedná o tabulku, ale uloženou proceduru. Systémová procedura *sp_describe_first_result_set* má jeden parametr, a to název procedury, kterou chce uživatel popsat.

Tato procedura vrátí velké množství sloupců, v aplikaci však používám pouze 3 z nich, a to *name*, *system_type_id* a *is_nullable*. Důvod je stejný jako u parametrů procedur, a to ten, že je nutné správně otypovat v C# i TypeScriptu všechny členy včetně informací, zda mohou mít hodnotu *null* či ne. Definici procedury je opět možné vidět v příloze 7, její zpracování je k naleznutí v příloze 12 a popis jednoho výstupu je k vidění na obrázku 30.



Obrázek 30: Výsledek získání výstupů procedury (Zdroj: vlastní zpracování)

3.2.9 Shrnutí přístupu k databázi

V této podkapitole jsem se věnoval přístupu k databázi z Node.js. Využil jsem celkem 5 systémových dotazů a 1 systémovou proceduru, díky kterým jsem schopný

popsat vstup a výstup procedury, namapovat jejich typy, a získat všechny databáze, schémata a procedury. Tato volání jsem také implementoval v Node.js za využití modulu *mssql*. Výstupem je 5 příkazů, které umožňují získat seznam výše zmíněných databázových objektu: Tyto příkazy jsou:

- listDb – vypíše seznam databází
- listSchema – vypíše seznam schémat
- listSp – vypíše seznam procedur v určené databázi a schématu
- listSpInputs – vypíše seznam vstupů procedury
- listSpOutputs – vypíše seznam výstupů procedury

Zejména poslední dva příkazy budou v nástroji důležité, neboť pomocí nich jsme schopni vygenerovat variabilní vstupy a výstupy procedur, respektive C# modelů, což je hlavním cílem diplomové práce.

3.3 Vytváření souborů v systému

V této podkapitole se zabývám vytvářením souborů na základě dat z příkazové řádky. V první části se zabývám vytvářením 6 souborů v systému, a to TypeScriptového *interface*, C# *modelu*, C# *DataAccess* třídy a C# *Controller* třídy. *Interface* a *model* jsou vytvořené dvakrát každý – jeden pro vstupní parametry, druhý pak pro výstupní parametry.

Pro vytváření souborů využívám dříve zmíněnou knihovnu *fs-extra*, která umožňuje pracovat se souborovým systémem. Celkem využívám 6 funkcí z této knihovny.

Funkce *existsSync* umožňuje zjistit, zda soubor existuje. Tuto funkcionalitu využívám zejména při čtení konfiguračních souborů a vytváření souborů. Při čtení konfiguračních souborů zjišťuji, zda konfigurační soubor existuje, a pokud ano, je možné ho přečíst. V opačném případě je nutné jej vytvořit. Při vytváření souborů jej využívám k tomu, abych byl schopný staré soubory smazat před tím, než vygeneruji nové.

Pro čtení souborů umožňuje Node.js využít funkci *readFileSync*, které umožní získat data ze souboru synchronně. Vzhledem k tomu, že čtení souborů používám pouze pro získání konfiguračních dat, je velmi důležité, aby se toto čtení provedlo synchronně,

jinak hrozí, že soubor bude generovaný před získáním konfiguračních parametrů a program může spadnout.

Zapisování do souborů skrze funkci *writeFileSync* také využívám pouze v rámci konfiguračních souborů a z výše zmíněných důvodu se musí také jednat o synchronní volání.

Předposlední funkcí je *unlinkSync*. Tato funkce slouží ke smazání souboru v systému a také je nutné v rámci této práce ji volat synchronně. Soubory totiž mažu předtím, než vytvářím nové, a v případě asynchronní metody hrozí kolize.

Nejčastěji používanou funkcí v této aplikaci je *appendFile*. Jak je možné vidět z absence slova *sync* v názvu, tato funkce je na rozdíl od ostatních asynchronní. Volá se pouze v případě, kdy se soubory opravdu mají vytvořit, a všechny ostatní kontroly jsou hotové, je tak možné generovat více souborů zaráz a ušetřit čas. Tuto funkci používám pro vygenerování každého z výše zmíněných souborů.

3.3.1 Mapování typů

Vzhledem k tomu, že v práci generuji typové soubory, je nutné brát v potaz to, že databázové typy mohou být mírně odlišné od C#. TypeScriptové typy jsou sice závislé na C# typech, nicméně se neřeší jejich striktnost. Zatímco v C# je více číselných typů, TS má pouze *number*. Příkladem rozdílu mezi databází a C# mohou být pak datové typy – zatímco databáze obsahuje typy jako *Date* a *DateTime*, C# zná pouze *DateTime*. Z toho důvodu je nutné vytvořit jejich mapování. Typy a jejich mapování je možné nalézt v příloze 13.

3.3.2 Vytvoření TS interface

Vytvoření TypeScriptového *interface* je velmi jednoduché. Přestože vytváříme 2 soubory, je možné pro oba soubory využít stejný kód, pouze s jinými parametry.

Parametry pro tento soubor je možné získat skrze již napsané funkce. V předchozí kapitole jsem vytvořil funkce, které získají popis vstupů a výstupů z databáze. Tyto funkce je nutné zavolat před vytvořením souboru. První soubor bude obsahovat vstupní parametry, druhý pak popisuje výstup procedury.

Po získání těchto parametrů je pak nutné pouze definovat cílový soubor, který chceme vytvořit. Pokud již soubor existuje, před jeho vytvořením bude smazán za využití funkce *unlink*. V opačném případě bude soubor pouze vytvořen.

Zdrojový kód pro vytvoření všech souborů je možné najít v příloze 14 a bude referována i v dalších částech této kapitoly. Příloha 15 pak obsahuje zdrojový kód pro získání obsahu souboru, který bude vytvořen.

Po zavolání příkazu *generate* a definovanou procedurou se vytvoří 2 soubory, pokud jsou definované vstupy i výstupy. V případě mého volání se vytvoří pouze soubor *outputInterface.ts*, neboť volám proceduru *fetch_main_data*, která neobsahuje žádné vstupní parametry. Tento soubor je možné vidět na obrázku 31.

```
export interface outputInterface {
    id: number;
    description_name?: string;
    when_joined?: Date;
    bank_status?: number;
    is_insured?: boolean;
}
```

Obrázek 31: Vygenerovaný TS interface (Zdroj: vlastní zpracování)

V případě, že bude zavolaná procedura, která má pouze vstupy a nemá žádný výstup, vytvoří se opět pouze jediný soubor, a to *inputInterface.ts*.

3.3.3 Vytvoření C# modelu

Pro vytvoření C# modelu je možné použít téměř identický algoritmus, jako pro vytvoření posledního souboru. Opět se jedná pouze o seznam proměnných a jejich typ, jediný rozdíl pro model je ve struktuře.

Typování v C# modelu je řešeno třídou namísto *interface*, a z toho důvodu bude pouze rozdíl v terminologii. Zároveň má C# více typů, a je nutné u nich být striktnější.

Pro vytvoření souboru je možné kód najít opět v příloze 14. Zdrojový kód pro získání obsahu souboru je pak k nalezení v příloze 16. Příklad vygenerovaného modelu je možné také vidět na obrázku 32.

```
public class outputModel {
    public int id { get; set; }
    public string description_name { get; set; }
    public DateTime? when_joined { get; set; }
    public double? bank_status { get; set; }
    public bool? is_insured { get; set; }
}
```

Obrázek 32: Vygenerovaný C# model (Zdroj: vlastní zpracování)

3.3.4 Vytvoření DataAccess třídy

V této podkapitole se zabývám vytvářením souboru DataAccess třídy. Tato třída je významně odlišná od předchozích vygenerovaných souborů.

Třída DataAccess se skládá z metody a členu třídy. Člen třídy je *CommandDefinition*, který slouží k definici uložené procedury. Tento člen má vždy stejnou strukturu – schéma a jméno uložené procedury v řetězcovém zápisu a definici všech vstupů. Jedná-li se tedy o proceduru bez vstupů, tento člen obsahuje pouze název uložené procedury. V opačném případě obsahuje název procedury a výpis všech parametrů procedury včetně jejich databázového typu.

Pro vytvoření tohoto členu využívám opět mapování a zdrojový kód je možné nalézt v příloze 17. Výsledek je možné vidět na obrázku na konci této kapitoly, který obsahuje finální vygenerovaný soubor.

Co se týče metody třídy, v této části již implementuji různé možnosti specifikace uživatele kromě jména procedury. Jak jsem zmínil v kapitole 2, současný generátor společnosti LOGEX využívá specifické metody pro volání procedury.

Při generování souborů je nutné brát v potaz metody, které uživatel specifikuje. Například pro metodu *ExecuteNonQuery* je nutné mít návratový typ *void*, neboť uložená procedura nevrací žádný výsledek, pouze se provede. Příkladem může být například přidání záznamu do tabulky, kdy žádný výsledek není očekávaný. Naopak při dotazování dat je nutné využít jednu z pěti jiných možností, které počítají s návratovými hodnotami.

V rámci samotného volání je rozdíl ve výstupních hodnotách a způsobu zavolání procedury. Při generování obsahu je tak nutné dělat podmínky, například zda se má funkce pouze provést (vkládání záznamů), anebo zda je očekávaná návratová hodnota

(dotaz na data). V případě, že se návratová hodnota očekává, je nutné definovat návratový typ.

V momentě, kdy máme definovanou návratovou hodnotu, uloženou proceduru a způsob volání, zbývá pouze jeden bod, a to definice argumentů metody. Většinu z těchto argumentů lze velmi snadno získat stejným způsobem, jako byl vytvoření člen *CommandDefinition*, tedy iterací nad vstupními parametry procedury.

V případě asynchronních metod však je posílán další parametr *CancellationToken*, který umožňuje přerušení volání. Tento parametr je automaticky vyžadován v případě využití asynchronních metod a musí být přidán do seznamu parametrů.

Posledním kondicionálním parametrem je *Guid*. Tento parametr slouží k revizi hodnot v cache. Každá hodnota uložená do cache má definovaný *Guid*, který slouží jako unikátní identifikátor. V případě, že tento *Guid* již existuje, hodnoty se vrátí z cache a nezatěžují výkon databáze. Pokud neexistuje nebo není validní, procedura se v databázi zavolá. Tento parametr se využívá pouze u metody *ExecuteToCacheAsync*.

Výsledek vygenerování DataAccess třídy je možné vidět na obrázku 33. Třída byla vytvořena metodou *ExecuteToCacheAsync* a má tak návratovou hodnotu.

```
public class DataAccess {

    public Task<ICacheableData<IDbMultiResult<modelName>>> fetch_main_data(
        Guid? cacheRevision,
        CancellationToken cancellationToken
    )
    {
        return fetch_main_data.ExecuteToCacheAsync<modelName>(
            cacheRevision,
            cancellationToken
        );
    }

    private static CommandDefinition fetch_main_data = CommandDefinition.DefineSp({
        "ui.fetch_main_data"
    );
}
```

Obrázek 33: Vygenerovaný DataAccess cache metodou (Zdroj: vlastní zpracování)

Pro lepší ilustraci rozdílu mezi metodou s návratovou hodnotou a vkládáním hodnot je na obrázku 34 možné vidět volání jiné procedury. V tomto případě však jsou

data vkládány do databáze, a tak definice procedury vyžaduje další parametry. Také metoda třídy již neobsahuje *Guid* a token.

```
public class DataAccess {

    public void add_main_data(
        int id
        string description_name
        DateTime when_joined
        double bank_status
        bool is_insured
    )
    {
        add_main_data.ExecuteNonQuery(
            id
            description_name
            when_joined
            bank_status
            is_insured
        );
    }

    private static CommandDefinition add_main_data = CommandDefinition.DefineSp({
        "ui.add_main_data"
        "@id", SqlDbType.Int
        "@description_name", SqlDbType.VarChar, 255,
        "@when_joined", SqlDbType.DateTime
        "@bank_status", SqlDbType.Float
        "@is_insured", SqlDbType.Bit
    );
}
```

Obrázek 34: Vygenerovaný DataAccess bez dotazu (Zdroj: vlastní zpracování)

Zdrojový kód pro vytvoření DataAccess třídy je možné nalézt v příloze 17 a 18, které obsahují vytváření obsahu souboru. Samotné vytvoření obsahu je možné nalézt v příloze 14.

3.3.5 Vytvoření Controller třídy

Posledním souborem, který je nutné vygenerovat, je *Controller*. Tento soubor slouží pro zachytávání http žádostí a vrácení dat. Příkladem může být například dotaz pro získání dat z databáze, který není možné dát na front-end aplikace. Z toho důvodu z front-endu zavoláme serverovou část, která validuje požadavek a poté vrátí data.

Pro sestavení *controlleru* využívám strukturu ASP.NET Core frameworku od společnosti Microsoft. Konkrétně využívám 3 atributy – *ApiController*, *Route* a *Authorize*. Tyto atributy není nutné definovat striktně – stačí využít pouze dočasné hodnoty určené k přepsání za využití komentářů.

Co se týče struktury Controlleru, platí zde stejné omezení, jako u Data Access třídy. To znamená, že musíme vracet rozdílné hodnoty na základě různých vybraných metod. Vzhledem k tomu, že tohle je již vyřešený problém, je možné znovu využít již vytvořené funkce. To samé platí o *CancellationToken* a *Guid* hodnotách.

V controlleru je také důležitý typ http dotazu. V případě dotazu *GET* je nutné totiž brát hodnoty z adresy za využití atributu *FromQuery*, zatímco při dotazu typu *POST* se posílá na controller objekt.

Na obrázku 35 je možné vidět vygenerovaný controller pro proceduru *add_main_data*, která vkládá data do databáze za využití metody POST a využití objektu.

```
[ApiController]
[Route(/* Controller Route Path */)]
[Authorize(/*Roles = ApplicationPermissions.Access*/)]
public class Controller: ApiController {

    [Route(/* Method Route Path */)]
    [Authorize(/* Roles = ApplicationPermissions.Edit */)]
    [HttpPost]
    public dynamic controllerPath(inputModel inputArguments)
    {
        add_main_data.ExecuteNonQuery(
            inputModel.id
            inputModel.description_name
            inputModel.when_joined
            inputModel.bank_status
            inputModel.is_insured
        );
    }
}
```

Obrázek 35: Vygenerovaný controller pro vkládání dat (Zdroj: vlastní zpracování)

Jak je možné vidět, metoda post definuje na vygenerovaném souboru atribut před definicí funkce a argumenty funkce. Samotné provedení procedury je také mírně odlišné, protože je nutné se odkazovat na objekt. Obrázek 36 obsahuje vygenerovaný controller pro získání dat metodou GET, která má mírně odlišnou strukturu.

```
[ApiController]
[Route(/* Controller Route Path */)]
[Authorize(/*Roles = ApplicationPermissions.Access*/)]
public class Controller: ApiController {

    [Route(/* Method Route Path */)]
    [Authorize(/* Roles = ApplicationPermissions.Edit */)]
    [HttpGet]
    public async Task<outputModel> controllerPath(
        Guid? cacheRevision,
        CancellationToken cancellationToken
    )
    {
        return fetch_main_data.ExecuteToCacheAsync<outputModel>(
            cacheRevision,
            cancellationToken
        );
    }
}
```

Obrázek 36: Vygenerovaný controller pro získání dat (Zdroj: vlastní zpracování)

Hlavním rozdílem jsou tak parametry funkce *controllerPath*, kde při metodě *POST* se všechny vyskytují jako objekt, zatímco u metody *GET* se jedná o seznam po sobě jdoucích hodnot. Zdrojový kód pro vytvoření obsahu je možné nalézt v příloze 19 a samotné vytvoření je opět součástí přílohy 14.

3.3.6 Shrnutí generování souborů

V této podkapitole jsem se zabýval generováním souborů skrze prostředí příkazové řádky. Všechny obrázky, které jsou v této části využité, obsahují výsledky generování z aplikace. Doposud však byly všechny parametry dodané uvnitř programu. Uživatel má možnost vygenerovat celkem 6 souborů.

3.4 Parametrizace příkazové řádky

Nyní lze říct, že současná aplikace již splňuje většinu požadavků. Práce nyní generuje soubory přímo v počítači, umožňuje všechny důležité metody generování definované společností LOGEX, a výsledek je také jasný.

Nicméně v současném stavu není možné definovat žádné metody. Pro předchozí příklady byly využité statické názvy databáze, schématu i procedur. Pokud chceme, aby uživatel mohl definovat vlastní parametry, musíme mu umožnit tuto funkcionalitu.

Pro tuto parametrizaci využívám knihovnu *yargs*. Přestože samotný node.js umožňuje přistupovat k parametrům skrze *process.argv*, museli bychom pro každý parametr napsat funkci. Tohle je problém, která právě knihovna řeší.

V knihovně můžeme definovat příkazy a možnosti. V diplomové práci používám pouze tři funkce. První z nich je funkce *command*, která definuje různé příkazy, které se budou volat. V diplomové práci využívám tento příkaz například pro odlišení generování souborů a konfiguraci globálních parametrů.

Pro každý příkaz můžeme dodat možnost *option*. Tato možnost definuje dodatečné možnosti, které se příkazu pošlou. Příkladem možnosti může být například databáze, na které chceme příkaz vykonat.

Poslední funkcí, který v práci používám, je *alias*. Tato funkce je spjatá s možností – každá možnost má jméno, a díky *alias* jsme schopní zkrátit zápis a využívat více možností.

Při vytváření programu jsem umožnil celkem 8 příkazů, z toho pět pro vypsání databázových objektů. Poslední tři příkazy jsou *config* pro konfiguraci aplikace, *generate* pro generování souborů, a *regenerate* pro opětovné vygenerování souborů se stejnými parametry.

3.4.1 Získání globální parametrizace

Globální parametrizace definuje defaultní hodnoty pro volání generujících funkcí. Tato konfigurace existuje jako soubor typu JSON a je možné jí získat či měnit příkazem *config*.

V případě, že chce uživatel získat konfiguraci, je možné využít příkaz *config get*, který získá veškerou konfiguraci pro program. Pokud chce získat konkrétní nastavení, například defaultní hodnoty pro funkce, může dodat specifický *namespace*. Pokud *namespace* neexistuje, vrátí se chybová hláška, kterou je možné vidět na obrázku 37.

Obrázek 37: Získání neexistující konfigurace (Zdroj: vlastní zpracování)

V případě, že konfigurace existuje, příkaz vypíše list uložených možností. Defaultní nastavení, která jsou v *namespace global*, je možné vidět na obrázku 38.

```
$ ts-node index.ts config get --ns="global"
{
  server: 'localhost\\SQLEXPRESS01',
  database: 'DataAccessGenerator_v2',
  schema: 'ui',
  generateInterface: true,
  generateModel: true,
  generateDataAccess: true,
  generateController: true
}
```

Obrázek 38: Získání defaultní parametrizace (Zdroj: vlastní zpracování)

3.4.2 Ukládání globální parametrizace

V případě, že uživatel chce změnit globální konfiguraci, je možné využít příkaz *config set*. Tento příkaz umožňuje seznam možností, které je možné specifikovat a uložit. Kompletní seznam možností je:

- Server – definuje defaultní server pro volání
- Database – definuje defaultní databázi pro volání
- Schema – definuje defaultní schema pro volání
- callType – definuje způsob volání. Interní metoda společnosti.
- generateInterface – definuje, zda se má vygenerovat TS interface
- generateModel – definuje, zda se má vygenerovat C# model
- generateDataAccess – definuje, zda se má vygenerovat C# DataAccess
- generateController – definuje, zda se má vygenerovat Controller
- dataAccessPath – definuje cestu, kde se má vytvořit dataAccess
- controllerPath – definuje cestu, kde se má vytvořit controller

Zdrojový kód implementace je možné vidět v příloze 20.

3.4.3 Lokální parametrizace příkazové řádky

Lokální parametrizace umožňuje využít stejné příkazy jako globální. V případě, že je lokální parametrizace definovaná, zavolá se příkaz s lokálními parametry. Pokud tyto parametry nejsou definované, program se je pokusí nahradit výše zmíněnými globálními parametry. Lokální parametrizace má 3 možnosti navíc:

- storedProcedureName – jméno procedury pro generování souborů
- route – URL cesta pro controller
- httpMethodType – http metoda (GET vs POST), která se pro volání využívá

Všechny tyto parametry definují strukturu vygenerovaných souborů a jejich obsah. Ukládání této konfigurace probíhá automaticky po každém volání tak, aby bylo možné v budoucnu zavolat opětovné generování souborů, které je možné využít v případě, že se databázová funkcionalita obohatila o jeden vstupní parametr a vše ostatní zůstává stejné. Zdrojový kód implementace parametrů příkazové řádky je možné nalézt v příloze 20.Tuto funkcionalitu řeším v následující kapitole.

3.5 Opětovné generování souborů

Opětovné generování souborů je velmi jednoduché. Vzhledem k tomu, že všechny funkce pro generování již byly vytvořené a dá se jim poslat parametry specifikované uživatelem, není potřeba dělat mnoho změn.

V podkapitole 3.4.3 jsem se zabýval lokální parametrizací, ve které jsem zmínil, že parametry se ukládají na konci každého generování souborů. Vzhledem k tomu, že tyto parametry jsou již uložené, stačí si uložit hodnotu z konfiguračního souboru do programu a zavolat již vytvořenou funkci pro generování souborů s těmito parametry. Zdrojový kód je možné vidět v příloze 20.

3.6 Veřejné vystavení programu

V tento moment je již aplikace příkazové řádky hotová. Aby však mohla být využívaná i z jiných počítačů a dalo se ji nainstalovat, je nutné ji zveřejnit do registru modulů.

3.6.1 Úpravy ve zdrojovém kódu

Pro zpřístupnění zdrojového kódu je nutné upravit pár konfiguračních souborů. Jak jsem již zmínil v první kapitole, počítače nejsou schopné interpretovat TypeScript samostatně. Z toho důvodu je nutné využít příkaz *tsc* pro transpilaci zdrojového kódu a tento proces vyžaduje soubor *tsconfig.json*, který také nalezneme v příloze 21.

Dále je nutné mírně upravit soubor *package.json*. V tomto souboru je nutné specifikovat jméno projektu, v tomto případě *dt-lgx-dafg*. Nakonec je potřeba specifikovat *bin* a *main* parametry, které definuji spustitelný soubor. Parametr *bin* musí již být transpilovaný soubor. V této aplikaci se jedná o *index.ts* pro *main* parametr, respektive *lib/index.js* pro *bin*. Tento soubor lze nalézt v příloze 22.

3.6.2 Zveřejnění programu

Pro zveřejnění programu je nutné mít účty na dvou platformách. První z nich je GitHub pro uložení zdrojového kódu. Druhá platforma se jmenuje npmjs a jedná se o správce balíčků pro JavaScript. Po přihlášení do obou platforem je možné spustit pouze příkaz *npm publish* ve složce, která obsahuje *package.json*, a poté je možné si balíček stáhnout skrze nástroj *npm*. Na obrázku 39 je možné vidět volání programu skrze jeho název *dt-lgx-dafg* a toto volání je možné provést z jakékoli složky počítače.

Obrázek 39: Volání globálně nainstalovaného balíčku dt-lgx-dafg (Zdroj: vlastní zpracování)

3.7 Shrnutí vlastního návrhu řešení

V poslední kapitole své diplomové práce jsem se věnoval vývoji aplikace, která slouží jako generátor souborů skrze prostředí příkazové řádky. Aplikace byla vyvinuta jazykem TypeScript za využití Node.js, MSSQL a pomocných knihoven.

Uživatel aplikace má možnost vygenerovat celkem 6 souborů, z toho 2 TypeScriptové *interface*, 2 C# *modely*, Controller a DataAccess. Uživatel má možnost specifikovat skrze konfigurační soubor *JSON* anebo jako parametr generátoru, zda chce vytvořit všechny soubory či nikoli.

Aplikace obsahuje 3 různé příkazy. První z těchto příkazů je *generate*, který slouží k samotnému generování souborů. Obsahuje 13 možností konfigurace, které jsou:

- Server – definuje defaultní server pro volání
- Database – definuje defaultní databázi pro volání
- Schema – definuje defaultní schema pro volání
- callType – definuje způsob volání. Interní metoda společnosti.
- generateInterface – definuje, zda se má vygenerovat TS interface
- generateModel – definuje, zda se má vygenerovat C# model
- generateDataAccess – definuje, zda se má vygenerovat C# DataAccess
- generateController – definuje, zda se má vygenerovat Controllre
- dataAccessPath – definuje cestu, kde se má vytvořit dataAccess
- controllerPath – definuje cestu, kde se má vytvořit controller
- storedProcedureName – jméno procedury pro generování souborů
- route – URL cesta pro controller
- httpMethodType – http metoda (GET vs POST), která se pro volání využívá

Všechny kromě posledních tří metod lze nakonfigurovat globálně pro každé volání, můžeme tak specifikovat většinu parametrů tak, abychom je nemuseli vyplňovat a měnili je pouze podle potřeby.

Druhým příkazem je *regenerate*, který po předání názvu procedury je schopný znovu vygenerovat již dříve vygenerovaný soubor. Tento příkaz lze využít v případě, že soubor byl již vygenerován, ale databáze změnila vstupní či výstupní parametry. Není tak nutné znovu specifikovat stejné parametry.

Posledním příkazem je *config*. Tento příkaz umožňuje skrze možnost *get* získat celý konfigurační soubor, případně jeho část s parametrem *namespace*. Další možností je *set*, čímž má uživatel možnost globálně nastavit parametry pro všechna volání. Seznam parametrů lze najít na předchozí stránce.

Aplikace je přístupná na portále *npmjs* a lze ji nainstalovat skrze správce balíčků *npm*. V případě globální instalace balíčku je možné volat příkaz *dt-lgx-dafg* z jakéhokoli umístění.

Vývoj aplikace trval celkem přibližně 4 měsíce a je možné jej rozdělit do následujících částí:

- Sběr požadavků a návrh aplikace
- Konstrukce aplikace
- Testování a ladění chyb

První část trvala nejkratší dobu a to přibližně 2 týdny, ve kterých jsem získával veškeré požadavky od společnosti a konzultoval možné přístupy. Konstrukce a příprava funkčního programu zabrala dalších 6 týdnů. O 2 týdny delší byla finální fáze testování a oprava chyb, kde největším problémem byly nepřehledné chybové hlášky a zachytávání krajních případů, se kterými se nedalo v návrhu počítat.

3.8 Přínos práce

V diplomové práci jsem dosáhl kvalitního základu pro automatizaci přístupu k databázovým strukturám. Podařilo dosáhnout všech cílů práce, splňuje veškeré požadavky společnosti, a značně zjednodušuje práci vůči předchozímu nástroji. Z toho důvodu věřím, že nástroj má vysoký potenciál na usnadnění práce vývojářům. Je však možné, že budou zapotřebí mírné úpravy a vylepšení nástroje, aby bylo možné zachytit všechny krajní případy použití, neboť příklady, na kterých jsem nástroj testoval, jsou velmi základní.

Přínos z ekonomického hlediska je velmi obtížné stanovit, a to zejména z důvodu velmi specifického využití. Jedná se však o automatizaci práce, která je nedílnou součástí každodenního vývoje. Přestože aplikace není zisková, umožňuje vývojářům věnovat se důležitějším věcem než repetitivním úkonům, které netvoří žádnou hodnotu, a mohou tak

trávit více času na vývoji nových funkcí. Z toho důvodu lze předpokládat, že tato automatizace v dlouhém období povede k vyšší efektivitě zaměstnanců.

ZÁVĚR

Výsledkem diplomové práce je aplikace v prostředí příkazové řádky, která slouží ke generování souborů pro přístup do databáze. Na základě přístupu k internímu nástroji společnosti jsem udělal analýzu současného stavu, ze kterého vzešly následující požadavky pro aplikaci:

- Vylepšení současných generátorových parametrů
- Změna výsledku volaných metod pro lepší srozumitelnost
- Vytváření souborů v počítači

Poslední z požadavků byl vyřešen samotným vývojem aplikace, protože díky příkazové řádce je možnost přístupu k souborovému systému. První dva požadavky jsem bral v potaz při jejím vývoji a snažil se udělat nástroj co nejvíc intuitivní. Kvůli tomu jsem také analyzoval jiný CLI nástroj, konkrétně *Angular CLI*, na jehož základě jsem definoval možnosti parametrizace.

Vývoj aplikace byl úspěšný a podařilo se mi balíček několikrát otestovat i ve firemním prostředí, zatím s kladným výsledkem. V případě větších zásahů do serverové či databázové části existuje riziko, že by bylo nutné aplikaci poupravit. Případné řešení by mohl řešit dodatečný vývoj této aplikace, který by umožnil generovat soubory pro různé jazyky, frameworky či databáze.

Při zpracování tématu se mi podařilo dosáhnout všech cílů, které jsem definoval po provedení analýzy. Aplikace je napsána za využití Node.js v jazyce TypeScript a je zdrojový kód je veřejně dostupný na GitHub repositáři, kde transpilovaný kód je ve složce *lib* (38). Balíček je také zveřejněný na software registru npmjs a lze jej stáhnout do počítače skrze správce balíčků *Node Package Manager* s příkazem *npm i dt-lgx-dafg*. Při globálním stažení balíčku je možné generovat soubory skrze příkaz *dt-lgx-dafg generate* v jakémkoli adresáři, je však nutné nejprve definovat zdrojovou databázi přes příkaz *dt-lgx-dafg config*. (39)

SEZNAM POUŽITÝCH ZDROJŮ

1. What's the difference between a console, a terminal, and a shell? *Scott Hanselman: Coder, Blogger, Teacher, Speaker, Author* [online]. c2021, September 12, 2019 [cit. 2021-03-18]. Dostupné z: https://www.hanselman.com/blog/whats-the-difference-between-a-console-a-terminal-and-a-shell
2. Telex machine TTY. In: *Flickr* [online]. Flickr, June 12, 2010 [cit. 2021-03-18]. Dostupné z: https://www.flickr.com/photos/15587432@N02/4669611994
3. Windows commands: Microsoft docs. *Oficiální domovská stránka Microsoft* [online]. Microsoft, c2021, 29. 6. 2020 [cit. 2021-03-18]. Dostupné z: https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/windows-commands
4. Shell Command Language. *The Open Group Website* [online]. c1995-2021, 2018 [cit. 2021-03-18]. Dostupné z: https://pubs.opengroup.org/onlinepubs/9699919799/utilities/V3_chap02.html
5. Sh. *The Open Group Website* [online]. c1995-2021, 2004 [cit. 2021-03-18]. Dostupné z: https://pubs.opengroup.org/onlinepubs/009695399/utilities/sh.html
6. Ksh88. *Oracle: Integrated Cloud Applications and Platform Services* [online]. Oracle, c2021 [cit. 2021-03-18]. Dostupné z: https://docs.oracle.com/cd/E36784_01/html/E36870/ksh88-1.html
7. Bash. *The GNU Operating System and the Free Software Movement* [online]. Free Software Foundation, c1996-2021 [cit. 2021-03-18]. Dostupné z: https://www.gnu.org/software/bash/
8. PowerShell Documentation. *Oficiální domovská stránka Microsoft* [online]. Microsoft, c2021 [cit. 2021-03-18]. Dostupné z: https://docs.microsoft.com/en-us/powershell/
9. JavaScript: MDN. *MDN Web Docs* [online]. Mozilla, c2005-2021 [cit. 2021-03-18]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript
10. ECMA-262: Ecma International. *Ecma International* [online]. Geneva, June 2020 [cit. 2021-03-18]. Dostupné z: https://www.ecma-international.org/publications-and-standards/standards/ecma-262/
11. *Can I use...: Support tables for HTML5, CSS3, etc.* [online]. [cit. 2021-03-18]. Dostupné z: https://caniuse.com/
12. Polyfill. *MDN Web Docs* [online]. Mozilla, c2005-2021 [cit. 2021-03-18]. Dostupné z: https://developer.mozilla.org/en-US/docs/Glossary/Polyfill
13. *TypeScript: Typed JavaScript at Any Scale* [online]. Microsoft, c2012-2021 [cit. 2021-03-18]. Dostupné z: https://www.typescriptlang.org/
14. *Node.js* [online]. OpenJS Foundation [cit. 2021-03-18]. Dostupné z: https://nodejs.org/en/
15. *Deno: A secure runtime for JavaScript and TypeScript* [online]. [cit. 2021-03-18]. Dostupné z: https://deno.land/

16. Sort("NODE") --> DENO. *DEV Community* [online]. c2016-2021, 7. 8. 2020 [cit. 2021-03-18]. Dostupné z: https://dev.to/nitdgplug/sort-node-deno-4nck
17. *Npm* [online]. npm, 2014 [cit. 2021-03-18]. Dostupné z: https://www.npmjs.com/
18. *Yarn - Package Manager: Home* [online]. c2016-2021 [cit. 2021-03-18]. Dostupné z: https://yarnpkg.com/
19. Asynchronous programming: Microsoft docs. *Oficiální domovská stránka Microsoft* [online]. Microsoft, c2021, 6. 4. 2020 [cit. 2021-03-18]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/
20. Concurrency model and the event loop: JavaScript. *MDN Web Docs* [online]. Mozilla, c2005-2021 [cit. 2021-03-18]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop
21. Web APIs: MDN. *MDN Web Docs* [online]. Mozilla, c2005-2021 [cit. 2021-03-18]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/API
22. Framework vs Library vs ... *Medium: Where good ideas find you* [online]. [cit. 2021-03-18]. Dostupné z: https://medium.com/@shashvatshukla/framework-vs-library-vs-platform-vs-api-vs-sdk-vs-toolkits-vs-ide-50a9473999db
23. Yargs. *Npm* [online]. npm, 2014 [cit. 2021-03-18]. Dostupné z: https://www.npmjs.com/package/yargs
24. Ts-node. *Npm* [online]. npm, 2014 [cit. 2021-03-18]. Dostupné z: https://www.npmjs.com/package/ts-node
25. Fs-extra. *Npm* [online]. npm, 2014 [cit. 2021-03-18]. Dostupné z: https://www.npmjs.com/package/fs-extra
26. Figlet. *Npm* [online]. npm, 2014 [cit. 2021-03-18]. Dostupné z: https://www.npmjs.com/package/figlet
27. Chalk. *Npm* [online]. npm, 2014 [cit. 2021-03-18]. Dostupné z: https://www.npmjs.com/package/chalk
28. Mssql. *Npm* [online]. npm, 2014 [cit. 2021-03-18]. Dostupné z: https://www.npmjs.com/package/mssql
29. Msnodesqlv8. *Npm* [online]. npm, 2014 [cit. 2021-03-18]. Dostupné z: https://www.npmjs.com/package/msnodesqlv8
30. Co je to databáze. *Oracle Česká Republika* [online]. Oracle, c2021 [cit. 2021-03-18]. Dostupné z: https://www.oracle.com/cz/database/what-is-database/
31. *MongoDB: The most popular database for modern apps* [online]. MongoDB, c2021 [cit. 2021-03-18]. Dostupné z: https://www.mongodb.com/
32. What is SQL Server. *SQL Server Tutorial* [online]. c2021 [cit. 2021-03-18]. Dostupné z: https://www.sqlservertutorial.net/getting-started/what-is-sql-server/
33. SQL Server Downloads. *Oficiální domovská stránka Microsoft* [online]. Microsoft, c2021 [cit. 2021-03-18]. Dostupné z: https://www.microsoft.com/en-us/sql-server/sql-server-downloads*Angular* [online]. Google, c2010-2021 [cit. 2021-03-18]. Dostupné z: https://angular.io/

34. *Angular* [online]. Google, c2010-2021 [cit. 2021-03-18]. Dostupné z: https://angular.io/
35. Flowchart Tutorial (with Symbols, Guide and Examples). *Visual Paradigm: Ideal Modeling & Diagramming for Agile Team Collaboration* [online]. Visual Paradigm, c2021 [cit. 2021-03-18]. Dostupné z: https://www.visual-paradigm.com/tutorials/flowchart-tutorial/
36. LOGEX Solution Center s.r.o. *Obchodní rejstřík firem* [online]. Kurzy.cz, c2000-2021 [cit. 2021-03-19]. Dostupné z: https://rejstrik-firem.kurzy.cz/06111904/logex-solution-center-sro/
37. LOGEX, V2H, Prodacapo and Ivbar join forces to turn ... *LOGEX Healthcare Analytics* [online]. LOGEX, c2021 [cit. 2021-03-19]. Dostupné z: https://logex.com/logex-offers-the-most-advanced-and-complete-solutions-in-healthcare-analytics/
38. PROCHÁZKA, Šimon. SimProch/diploma-thesis: Diploma thesis for Data Access Generator. *GitHub* [online]. GitHub, c2021 [cit. 2021-03-18]. Dostupné z: https://github.com/SimProch/diploma-thesis
39. PROCHÁZKA, Šimon. Dt-lgx-dafg: Data Access Generator. *Npm* [online]. npm, 2014 [cit. 2021-03-18]. Dostupné z: https://www.npmjs.com/package/dt-lgx-dafg

SEZNAM OBRÁZKŮ

SEZNAM PŘÍLOH

Příloha č. 1: Vytvoření databází, schémat a tabulek

```
use DataAccessGenerator_v2
go

drop table if exists Test_Table
drop table if exists dev.Test_Table_Second
drop schema if exists dev
go

create schema dev
go

create table Test_Table(
	id int primary key,
	description_name  varchar(255),
	when_joined DateTime,
	bank_status float,
	is_insured bit
)

create table dev.Test_Table_Second(
	id int primary key,
	description_name varchar(255),
	when_joined DateTime,
	bank_status float,
	is_insured bit
)

use DataAccessGeneratorDB

drop table if exists Test_Table
drop table if exists dev.Test_Table_Second
drop schema if exists dev
go

create schema dev
go

create table Test_Table(
	id int primary key,
	description_name  varchar(255),
	when_joined DateTime,
	bank_status float,
	is_insured bit
)

create table dev.Test_Table_Second(
	id int primary key,
	description_name varchar(255),
	when_joined DateTime,
	bank_status float,
	is_insured bit
)
```

Příloha č. 2: Vložení prvních hodnot do tabulek

```
insert into DataAccessGenerator_v2.dev.Test_Table_Second
       values (1, 'some name', GETDATE(), 200.5, 1)


insert into DataAccessGenerator_v2.dbo.Test_Table
       values (1, 'some name', GETDATE(), 200.5, 1)


insert into DataAccessGeneratorDB.dev.Test_Table_Second
       values (1, 'some name', GETDATE(), 200.5, 1)


insert into DataAccessGeneratorDB.dbo.Test_Table
       values (1, 'some name', GETDATE(), 200.5, 1)
```

Příloha č. 3: Vytvoření procedur pro získání dat

```
use DataAccessGeneratorDB
go
drop procedure if exists ui.fetch_main_data_second
drop procedure if exists ui.fetch_main_data
drop procedure if exists ui.add_main_data_second
drop procedure if exists ui.add_main_data
go
drop schema if exists ui
go
create schema ui
go

use DataAccessGenerator_v2
go
drop procedure if exists ui.fetch_main_data_second
drop procedure if exists ui.fetch_main_data
drop procedure if exists ui.add_main_data_second
drop procedure if exists ui.add_main_data
go
drop schema if exists ui
go
create schema ui
go

use DataAccessGeneratorDB
go
create procedure ui.fetch_main_data_second
as
select * from DataAccessGeneratorDB.dev.Test_Table_Second
go

create procedure ui.fetch_main_data
as
select * from DataAccessGeneratorDB.dbo.Test_Table
go


use DataAccessGenerator_v2
go
create procedure ui.fetch_main_data_second
as
select * from DataAccessGenerator_v2.dev.Test_Table_Second
go

create procedure ui.fetch_main_data
as
select * from DataAccessGenerator_v2.dbo.Test_Table
go
```

Příloha č. 4: Vytvoření procedur pro vložení dat

```
use DataAccessGeneratorDB
go
drop procedure if exists ui.add_main_data_second
drop procedure if exists ui.add_main_data
go

use DataAccessGenerator_v2
go
drop procedure if exists ui.add_main_data_second
drop procedure if exists ui.add_main_data
go

use DataAccessGeneratorDB
go
create procedure ui.add_main_data_second
@id int,
@description_name varchar(255),
@when_joined DateTime,
@bank_status float,
@is_insured bit
as
insert into DataAccessGeneratorDB.dev.Test_Table_Second
values (@id, @description_name, @when_joined, @bank_status, @is_insured)
go

create procedure ui.add_main_data
@id int,
@description_name varchar(255),
@when_joined DateTime,
@bank_status float,
@is_insured bit
as
insert into DataAccessGeneratorDB.dbo.Test_Table
values (@id, @description_name, @when_joined, @bank_status, @is_insured)
go


use DataAccessGenerator_v2
go
create procedure ui.add_main_data_second
@id int,
@description_name varchar(255),
@when_joined DateTime,
@bank_status float,
@is_insured bit
as
insert into DataAccessGenerator_v2.dev.Test_Table_Second
values (@id, @description_name, @when_joined, @bank_status, @is_insured)
go

create procedure ui.add_main_data
@id int,
@description_name varchar(255),
@when_joined DateTime,
@bank_status float,
@is_insured bit
as
insert into DataAccessGenerator_v2.dbo.Test_Table
```

```
values (@id, @description_name, @when_joined, @bank_status, @is_insured)
go
```

Příloha č. 5: Volání procedur

```
delete from DataAccessGenerator_v2.dev.Test_Table_Second where id = 2
delete from DataAccessGenerator_v2.dbo.Test_Table where id = 2
delete from DataAccessGeneratorDB.dev.Test_Table_Second where id = 2
delete from DataAccessGeneratorDB.dbo.Test_Table where id = 2

declare @tmp DATETIME
set @tmp = GETDATE()

exec DataAccessGenerator_v2.ui.add_main_data_second
                            @id = 2,
                            @description_name = 'Some other name',
                            @when_joined = @tmp,
                            @bank_status = 255.4,
                            @is_insured = 0;

exec DataAccessGenerator_v2.ui.add_main_data
                            @id = 2,
                            @description_name = 'Some other name',
                            @when_joined = @tmp,
                            @bank_status = 255.4,
                            @is_insured = 0;

exec DataAccessGeneratorDB.ui.add_main_data_second
                            @id = 2,
                            @description_name = 'Some other name',
                            @when_joined = @tmp,
                            @bank_status = 255.4,
                            @is_insured = 0;

exec DataAccessGeneratorDB.ui.add_main_data
                            @id = 2,
                            @description_name = 'Some other name',
                            @when_joined = @tmp,
                            @bank_status = 255.4,
                            @is_insured = 0;

exec DataAccessGenerator_v2.ui.fetch_main_data
exec DataAccessGenerator_v2.ui.fetch_main_data_second
exec DataAccessGeneratorDB.ui.fetch_main_data
exec DataAccessGeneratorDB.ui.fetch_main_data_second
```

Příloha č. 6: Připojení k databázi

```
let config: sql.config;

export function initSqlConfig(): void {
    config = {
        server: getConfigObject().global.server,
        database: getConfigObject().global.database,
        driver: "msnodesqlv8",
        options: {
            trustedConnection: true,
        },
    };
}

export function updateSqlConfig(server: string, database: string): void {
    if (server) config.server = server;
    if (database) config.database = database;
}

export async function tryConnect(): Promise<sql.ConnectionPool> {
    try {
        const pool = new sql.ConnectionPool(config);
        return await pool.connect()
    } catch (error) {
        throw new Error(error);
    }
}
```

Příloha č. 7: Databázové dotazy

```
export function getDatabaseListQuery(): string {
    return `SELECT name FROM master.sys.databases
    WHERE Cast(CASE WHEN name IN ('master', 'model', 'msdb', 'tempdb') TH
EN 1 ELSE is_distributor END As bit) = 0
   `;
}

export function getStoredProcedureListQuery(databaseName: string, schema?
: string): string {
    if (!schema) return `SELECT ROUTINE_NAME FROM ${databaseName}.INFORMA
TION_SCHEMA.ROUTINES WHERE ROUTINE_TYPE = 'PROCEDURE'`;
    return `SELECT ROUTINE_NAME FROM ${databaseName}.INFORMATION_SCHEMA.R
OUTINES WHERE ROUTINE_TYPE = 'PROCEDURE' AND ROUTINE_SCHEMA = '${schema}'
`;
```

```
}

export function getSchemaListQuery(databaseName: string): string {
    return `SELECT SCHEMA_NAME FROM ${databaseName}.INFORMATION_SCHEMA.SC
HEMATA
    where SCHEMA_NAME NOT LIKE 'db[_]%' AND SCHEMA_NAME NOT IN ('guest',
'INFORMATION_SCHEMA', 'sys')`;
}

export function getProcedureOutputListQuery(databaseName: string, schemaN
ame: string, storedProcedureName: string): string {
    return `EXEC sp_describe_first_result_set N'${databaseName}.${schemaN
ame}.${storedProcedureName}'`
}

export function getProcedureInputListQuery(databaseName: string, schemaNa
me: string, storedProcedureName: string): string {
    return `SELECT name, system_type_id, is_nullable, max_length FROM ${d
atabaseName}.sys.parameters WHERE object_id = object_id('${databaseName}.
${schemaName}.${storedProcedureName}')`
}

export function getDbTypeListQuery(databaseName): string {
    return `SELECT name, system_type_id FROM ${databaseName}.sys.types`;
}
```

Příloha č. 8: Výpis seznamu databází

```
export function listDatabases(): Promise<sql.IRecordSet<DatabaseQueryResu
lt>> {
    return tryConnect().then(doListDatabases)

    async function doListDatabases(pool: sql.ConnectionPool): Promise<sql
.IRecordSet<DatabaseQueryResult>> {
        const request = new sql.Request(pool);
        const result = (await request.query<DatabaseQueryResult>(getDatab
aseListQuery())).recordsets[0];
        return result;
    }
}
```

Příloha č. 9: Výpis seznamu schémat

```
export function listSchemas(databaseName: string): Promise<sql.IRecordSet
<SchemaQueryResult>> {
    if (!databaseName) throw new Error(chalk.red("This command requires a
 database specified"));
    return tryConnect().then(doListSchemas);

    async function doListSchemas(pool: sql.ConnectionPool): Promise<sql.I
RecordSet<SchemaQueryResult>> {
        const request = new sql.Request(pool);
        const result = (await request.query<SchemaQueryResult>(getSchemaL
istQuery(databaseName))).recordsets[0];
        return result;
    }
}
```

Příloha č. 10: Výpis seznamu procedur

```
export function listStoredProcedures(databaseName: string, schemaName: st
ring): Promise<sql.IRecordSet<StoredProcedureQueryResult>> {
    if (!databaseName) throw new Error(chalk.red("This command requires a
 database specified"));
    if (!schemaName) throw new Error(chalk.red("This command requires a s
chema specified"));
    return tryConnect().then(doListStoredProcedures);

    async function doListStoredProcedures(pool: sql.ConnectionPool): Prom
ise<sql.IRecordSet<StoredProcedureQueryResult>> {
        const request = new sql.Request(pool);
        const result = (await request.query<StoredProcedureQueryResult>(g
etStoredProcedureListQuery(databaseName, schemaName)))
            .recordsets[0];
        return result;
    }
}
```

Příloha č. 11: Výpis parametrů procedury

```
export function listProcedureInput(databaseName: string, schemaName: stri
ng, storedProcedureName: string): Promise<InputList> {
    if (!databaseName) throw new Error(chalk.red("This command requires a
 database specified"));
    if (!schemaName) throw new Error(chalk.red("This command requires a s
chema specified"));
    if (!storedProcedureName) throw new Error(chalk.red("This command req
uires a stored procedure name specified"));
    return tryConnect().then(doListProcedureInput);

    async function doListProcedureInput(pool: sql.ConnectionPool): Promis
e<InputList> {
        const request = new sql.Request(pool);
        const inputs = (await request.query<DbDescriptionType>(getProcedu
reInputListQuery(databaseName, schemaName, storedProcedureName)))
            .recordsets[0];
        const types = (await request.query<DbDescriptionType>(getDbTypeLi
stQuery(databaseName))).recordsets[0];
        const result: InputList = inputs.map((input) => {
            const variableName = types.find((type) => type.system_type_id
 === input.system_type_id).name as DBType;
            return {
                inputName: input.name,
                variableName: variableName,
                maxLength: input.max_length,
                isNullable: input.is_nullable,
            };
        });
        return result;
    }
}
```

Příloha č. 12: Výpis výstupů procedury

```
export function listProcedureOutput(databaseName: string, schemaName: str
ing, storedProcedureName: string): Promise<OutputList> {
    if (!databaseName) throw new Error(chalk.red("This command requires a
 database specified"));
    if (!schemaName) throw new Error(chalk.red("This command requires a s
chema specified"));
    if (!storedProcedureName) throw new Error(chalk.red("This command req
uires a stored procedure name specified"));
```

```
    return tryConnect().then(doListStoredProcedureOutput);

    async function doListStoredProcedureOutput(pool: sql.ConnectionPool):
 Promise<OutputList> {
        const request = new sql.Request(pool);
        const output = (await request.query<DbDescriptionType>(getProcedu
reOutputListQuery(databaseName, schemaName, storedProcedureName)))
            .recordsets[0];
        const types = (await request.query<DbDescriptionType>(getDbTypeLi
stQuery(databaseName))).recordsets[0];
        const result: OutputList = output.map((output) => {
            const variableName = types.find((type) => type.system_type_id
 === output.system_type_id).name as DBType;
            return {
                outputName: output.name,
                variableName: variableName,
                maxLength: output.max_length,
                isNullable: output.is_nullable,
            };
        });
        return result;
    }
}
```

Příloha č. 13: Typy a mapovací funkce

```
export type CSType = "string" | "int" | "Int16" | "Int64" | "byte" | "Gui
d" | "DateTime" | "double" | "decimal" | "bool";
export type TSType = "string" | "number" | "date" | "boolean";
export type DBType = keyof typeof DbToCsCommandDefinition;
export type DataAccessEnumType =
    | "BigInt"
    | "VarBinary"
    | "Bit"
    | "Char"
    | "DateTime"
    | "DateTime2"
    | "DateTimeOffset"
    | "Decimal"
    | "Float"
    | "Binary"
    | "Int"
    | "Money"
    | "NChar"
    | "NText"
```

```
    | "NVarChar"
    | "Real"
    | "Timestamp"
    | "SmallInt"
    | "SmallMoney"
    | "Variant"
    | "Text"
    | "Time"
    | "TinyInt"
    | "UniqueIdentifier"
    | "VarChar"
    | "Xml";

export interface CommandDefinitionProperties {
    propertyName: string;
    typeName: DataAccessEnumType;
    isNullable: boolean;
    maxLength: number;
}

export interface InterfaceProperties {
    propertyName: string;
    typeName: TSType;
    isNullable: boolean;
}

export interface ModelProperties {
    propertyName: string;
    typeName: CSType;
    isNullable: boolean;
}

export enum DbToCsharpModel {
    varchar = "string",
    nvarchar = "string",
    int = "int",
    smallint = "Int16",
    tinyint = "byte",
    bigint = "Int64",
    uniqueidentifier = "Guid",
    date = "DateTime",
    datetime = "DateTime",
    datetime2 = "DateTime",
    float = "double",
    decimal = "decimal",
    numeric = "decimal",
    money = "decimal",
```

```
    bit = "bool",
}

export enum DbToTsInterface {
    varchar = "string",
    nvarchar = "string",
    int = "number",
    smallint = "number",
    tinyint = "number",
    bigint = "number",
    uniqueidentifier = "string",
    date = "Date",
    datetime = "Date",
    datetime2 = "Date",
    float = "number",
    decimal = "number",
    numeric = "number",
    money = "number",
    bit = "boolean",
}

export enum DbToCsCommandDefinition {
    bigint = "BigInt",
    binary = "VarBinary",
    bit = "Bit",
    char = "Char",
    date = "DateTime",
    datetime = "DateTime",
    datetime2 = "DateTime2",
    datetimeoffset = "DateTimeOffset",
    decimal = "Decimal",
    float = "Float",
    image = "Binary",
    int = "Int",
    money = "Money",
    nchar = "NChar",
    ntext = "NText",
    numeric = "Decimal",
    nvarchar = "NVarChar",
    real = "Real",
    rowversion = "Timestamp",
    smalldatetime = "DateTime",
    smallint = "SmallInt",
    smallmoney = "SmallMoney",
    sql_variant = "Variant",
    text = "Text",
    time = "Time",
```

```
    timestamp = "Timestamp",
    tinyint = "TinyInt",
    uniqueidentifier = "UniqueIdentifier",
    varbinary = "VarBinary",
    varchar = "VarChar",
    xml = "Xml",
}

export function mapDbToCsCommandDefinition(key: DBType): string {
    return DbToCsCommandDefinition[key];
}
export function mapDbToTsTsInterface(key: DBType): string {
    return DbToTsInterface[key];
}
export function mapDbToCsModel(key: DBType): string {
    return DbToCsharpModel[key];
}
```

Příloha č. 14: Funkce pro vytvoření souborů

```
const LINE_END = "\r\n";
const rootDirectory = path.dirname(require.main.filename);

export function createTsInterface(interfaceName: string, props: Interface
Properties[]): Promise<string> {
    return new Promise((resolve, reject) => {
        const filePath = `${rootDirectory}/result/${interfaceName}.ts`;
        deleteFileIfExists(filePath);
        const getInterfaceArgs: getInterfaceArguments = {
            interfaceName: interfaceName,
            properties: props,
        };
        const newInterface = getInterface(getInterfaceArgs);
        fs.appendFile(filePath, newInterface, (err) => postCreateCallback
(err, filePath, resolve, reject));
    });
}
export function createCsModel(modelName: string, props: ModelProperties[]
): Promise<string> {
    return new Promise((resolve, reject) => {
        const filePath = `${rootDirectory}/result/${modelName}.cs`;
        deleteFileIfExists(filePath);
        const getModelArgs: getModelArguments = {
            modelName: modelName,
            properties: props,
```

```
        };
        const newModel = getModel(getModelArgs);
        fs.appendFile(filePath, newModel, (err) => postCreateCallback(err
, filePath, resolve, reject));
    });
}
export function createDataAccess(
    methodType: MethodCallType,
    schema: string,
    spName: string,
    props: CommandDefinitionProperties[],
    modelProps: ModelProperties[]
): Promise<string> {
    return new Promise((resolve, reject) => {
        const filePath = `${rootDirectory}/result/${spName}.cs`;
        deleteFileIfExists(filePath);
        const dataAccessArgs: getDataAccessArguments = {
            methodType: methodType,
            spName: spName,
            modelName: "modelName",
            properties: modelProps,
        };
        const commandDefinitionArgs: getCommandDefinitionArguments = {
            schema: schema,
            spName: spName,
            properties: props,
        };
        const dataAccessMethod = getDataAccess(dataAccessArgs);
        const commandDefinition = getCommandDefinition(commandDefinitionA
rgs);
        const spaceBetween = LINE_END + LINE_END;
        const fileStart = `public class DataAccess {${spaceBetween}`;
        const dataAccessContents = dataAccessMethod + spaceBetween;
        const commandDefinitionContents = commandDefinition + LINE_END;
        const fileEnd = `}`;
        const fileContents = fileStart + dataAccessContents + commandDefi
nitionContents + fileEnd;
        fs.appendFile(filePath, fileContents, (err) => postCreateCallback
(err, filePath, resolve, reject));
    });
}

export function createController(args: getControllerArguments): Promise<s
tring> {
    return new Promise((resolve, reject) => {
        const filePath = `${rootDirectory}/result/${args.classMethodName}
.cs`;
```

```
        deleteFileIfExists(filePath);
        const controller = getController(args);
        fs.appendFile(filePath, controller, (err) => postCreateCallback(e
rr, filePath, resolve, reject));
    });
}

function deleteFileIfExists(fileName: string): void {
    if (fs.existsSync(fileName)) fs.unlinkSync(fileName);
}
function postCreateCallback(err: NodeJS.ErrnoException, fileName: string,
 resolve: Function, reject: Function): void {
    if (err) reject(err);
    resolve(fileName);
}
```

Příloha č. 15: Funkce pro získání obsahu souboru interface

```
const LINE_END = "\r\n";

export default function getInterface(args: getInterfaceArguments): string
 {
    const interfaceStart = `export interface ${args.interfaceName} {${LIN
E_END}`;
    let typing = "";
    args.properties.forEach((property) => {
        typing += addTypeLine(property.propertyName, property.typeName, p
roperty.isNullable);
    });
    const interfaceEnd = `}`;
    const result = interfaceStart + typing + interfaceEnd;
    return result;
}

function addTypeLine(propertyName: string, typeName: TSType, isNullable:
boolean): string {
    const typingToAdd = `    ${propertyName}${isNullable ? "?" : ""}: ${t
ypeName};${LINE_END}`;
    return typingToAdd;
}
```

Příloha č. 16: Funkce pro získání obsahu souboru model

```
const LINE_END = "\r\n";

export default function getModel(args: getModelArguments): string {
    const modelStart = `public class ${args.modelName} {${LINE_END}`;
    let typing = "";
    args.properties.forEach((property) => {
        typing += addTypeLine(property.propertyName, property.typeName, p
roperty.isNullable);
    });
    const modelEnd = `}`;
    const result = modelStart + typing + modelEnd;
    return result;
}

function addTypeLine(propertyName: string, typeName: CSType, isNullable:
boolean): string {
    const typingToAdd = `    public ${typeName}${isNullable ? "?" : ""} $
{propertyName} { get; set; }${LINE_END}`;
    return typingToAdd;
}
```

Příloha č. 17: Funkce pro získání obsahu CommandDefinition

```
const LINE_END = "\r\n";

export default function getCommandDefinition(args: getCommandDefinitionAr
guments): string {
    const commandDefinitionStart = `    private static CommandDefinition
${args.spName} = CommandDefinition.DefineSp({${LINE_END}`;
    let typing = `        "${args.schema}.${args.spName}"${LINE_END}`;
    args.properties.forEach((property) => {
        typing += addTypeLine(property.propertyName, property.typeName, p
roperty.maxLength);
    });
    const commandDefinitionEnd = `    );`;
    const result = commandDefinitionStart + typing + commandDefinitionEnd
;
    return result;
}

function addTypeLine(propertyName: string, typeName: DataAccessEnumType,
maxLength: number): string {
```

```
    const specifyMaxLength = typeName === "NVarChar" || typeName === "Var
Char";
    const typingToAdd = `         "@${propertyName}", SqlDbType.${typeName
}${specifyMaxLength ? ", " + maxLength + "," : ""}${LINE_END}`;
    return typingToAdd;
}
```

Příloha č. 18: Funkce pro získání obsahu DataAccess

```
const LINE_END = "\r\n";
const asyncMethods: MethodCallType[] = ["ExecuteToObjectsAsync", "Execute
ToDynamicAsync", "ExecuteToCacheAsync"];

export default function getDataAccess(args: getDataAccessArguments): stri
ng {
    const resultType = getTypeBasedOnCallMethod(args.methodType, args.mod
elName);
    const methodStart = `    public ${resultType} ${args.spName}(${LINE_E
ND}`;
    const queryArgs = getQueryArguments(args.methodType, args.properties)
;
    const methodBody = getQueryBody(args.methodType, args.spName, args.mo
delName, args.properties);
    const methodEnd = `    }`;
    const result = methodStart + queryArgs + methodBody + methodEnd;
    return result;
}

function getTypeBasedOnCallMethod(methodType: MethodCallType, modelName:
string) {
    if (methodType === "ExecuteToCacheAsync") return `Task<ICacheableData
<IDbMultiResult<${modelName}>>>`;
    if (methodType === "ExecuteToObjects") return `Task<${modelName}>`;
    if (methodType === "ExecuteToObjectsAsync") return `Task<IEnumerable<
${modelName}>>`;
    if (methodType === "ExecuteToDynamic") return `Task<dynamic>`;
    if (methodType === "ExecuteToDynamicAsync") return `Task<IList<dynami
c>>`;
    if (methodType === "ExecuteNonQuery") return "void";
    throw new Error(chalk.red("Unknown call type!"));
}

function getQueryArguments(methodType: MethodCallType, properties: ModelP
roperties[]): string {
    let result = "";
```

```
    const hasGuid = methodType === "ExecuteToCacheAsync";
    const hasCancellation = asyncMethods.includes(methodType);
    properties.forEach((i) => {
        result += `        ${i.typeName} ${i.propertyName}${hasGuid || ha
sCancellation ? ", " : ""} ${LINE_END}`;
    });
    if (hasGuid) result += `        Guid? cacheRevision${hasCancellation
? ", " : ""} ${LINE_END}`;
    if (hasCancellation) result += `        CancellationToken cancellatio
nToken ${LINE_END}`;
    result += "    )" + LINE_END + "    {" + LINE_END;
    return result;
}

function getQueryBody(methodType: MethodCallType, spName: string, modelNa
me: string, properties: ModelProperties[]): string {
    const functionCall = getExecutionCall(methodType, spName, modelName);
    const params = getQueryParameters(methodType, properties);
    return functionCall + params;
}

function getExecutionCall(methodType: MethodCallType, spName: string, mod
elName: string): string {
    let result = `${spName}.${methodType}`;
    if (methodType === "ExecuteNonQuery") return `        ${result}`;
    result = "        return " + result;
    if (methodType === "ExecuteToDynamic" || methodType === "ExecuteToDyn
amicAsync") return result;
    result += `<${modelName}>`;
    return result;
}

function getQueryParameters(methodType: MethodCallType, properties: Model
Properties[]): string {
    let result = "(" + LINE_END;
    const hasGuid = methodType === "ExecuteToCacheAsync";
    const hasCancellation = asyncMethods.includes(methodType);
    properties.forEach((i) => {
        result += `            ${i.propertyName}${hasGuid || hasCancellat
ion ? ", " : ""} ${LINE_END}`;
    });
    if (hasGuid) result += `            cacheRevision${hasCancellation ?
", " : ""} ${LINE_END}`;
    if (hasCancellation) result += `            cancellationToken ${LINE_
END}`;
    result += "        );" + LINE_END;
    return result;
```

```
}
```

Příloha č. 19: Funkce pro získání obsahu souboru Controller

```
const LINE_END = "\r\n";
const asyncMethods: MethodCallType[] = ["ExecuteToObjectsAsync", "Execute
ToDynamicAsync", "ExecuteToCacheAsync"];
const typedMethods: MethodCallType[] = ["ExecuteToObjects", "ExecuteToObj
ectsAsync", "ExecuteToCacheAsync"];

export default function getController(args: getControllerArguments): stri
ng {
    const spaceBetween = LINE_END;
    const fileStart = getControllerClass(args.routePath);
    const method = getMethod(args);
    const fileEnd = `}`;
    const result = fileStart + spaceBetween + method + fileEnd;
    return result;
}

function getControllerClass(routePath: string) {
    const apiController = `[ApiController]${LINE_END}`;
    const route = `[Route(/* Controller Route Path */)]${LINE_END}`;
    const auth = `[Authorize(/*Roles = ApplicationPermissions.Access*/)]$
{LINE_END}`;
    const fileContents = `public class Controller: ApiController {${LINE_
END}`;
    const fileStart = apiController + route + auth + fileContents;
    return fileStart;
}

function getMethod(args: getControllerArguments): string {
    const returnType = getTypeBasedOnCallMethod(args.methodType, args.out
putModelName);
    const route = `    [Route(${args.routePath ? args.routePath : "/* Met
hod Route Path */"})]${LINE_END}`;
    const auth = `    [Authorize(/* Roles = ApplicationPermissions.Edit *
/)]${LINE_END}`;
    const request = `    [${args.requestType === "POST" ? "HttpPost" : "H
ttpGet"}]${LINE_END}`;
    const methodStart = `    public ${returnType} ${args.classMethodName}
(${LINE_END}`;
    const methodArguments = getMethodArguments(args.methodType, args.requ
estType, args.properties, args.inputModelName);
    const methodContents = getMethodContents(args);
```

```
    const methodEnd = `    }${LINE_END}`;
    const methodAttributes = route + auth + request;
    const method = methodStart + methodArguments + methodContents + metho
dEnd;
    const result = methodAttributes + method;
    return result;
}

function getTypeBasedOnCallMethod(methodType: MethodCallType, modelName:
string) {
    if (methodType === "ExecuteToCacheAsync") return `async Task<${modelN
ame}>`;
    if (methodType === "ExecuteToObjects") return `Task<${modelName}>`;
    if (methodType === "ExecuteToObjectsAsync") return `async Task<IEnume
rable<${modelName}>>`;
    if (methodType === "ExecuteToDynamic") return `Task<dynamic>`;
    if (methodType === "ExecuteToDynamicAsync") return `async Task<IList<
dynamic>>`;
    if (methodType === "ExecuteNonQuery") return "dynamic";
    throw new Error(chalk.red("Unknown call type!"));
}

function getMethodArguments(
    methodType: MethodCallType,
    requestType: "POST" | "GET",
    properties: ModelProperties[],
    inputPropertiesName: string
): string {
    let result = "";
    result += getStandardProperties(requestType, methodType, properties,
inputPropertiesName);
    result += getAsynchronousProperties(methodType);
    result += `    )${LINE_END}    {${LINE_END}`;
    return result;
}

function getStandardProperties(
    requestType: "POST" | "GET",
    methodType: MethodCallType,
    properties: ModelProperties[],
    inputPropertiesName: string
): string {
    if (requestType === "POST") return inputPropertiesName;
    const hasGuid = methodType === "ExecuteToCacheAsync";
    const hasCancellation = asyncMethods.includes(methodType);
    let result = "";
    properties.forEach((i) => {
```

```
        result += `        [FromURI] ${i.typeName} ${i.propertyName}${has
Guid || hasCancellation ? ", " : ""} ${LINE_END}`;
    });
    return result;
}

function getAsynchronousProperties(methodType: MethodCallType) {
    let result = "";
    const hasGuid = methodType === "ExecuteToCacheAsync";
    const hasCancellation = asyncMethods.includes(methodType);
    if (hasGuid) result += `        Guid? cacheRevision${hasCancellation
? ", " : ""} ${LINE_END}`;
    if (hasCancellation) result += `        CancellationToken cancellatio
nToken ${LINE_END}`;
    return result;
}

function getMethodContents(args: getControllerArguments): string {
    let result = "        return ";
    const hasType = typedMethods.includes(args.methodType);
    const generics = hasType ? `<${args.outputModelName}>` : "";
    const dataAccessCall = `${args.dataAccessName}.${args.methodType}${ge
nerics}(${LINE_END}`;
    const parameters = getDataAccessParameters(args.methodType, args.prop
erties);
    result += dataAccessCall;
    result += parameters;
    return result;
}

function getDataAccessParameters(methodType: MethodCallType, properties:
ModelProperties[]): string {
    let result = "";
    const hasGuid = methodType === "ExecuteToCacheAsync";
    const hasCancellation = asyncMethods.includes(methodType);
    properties.forEach((i) => {
        result += `            ${i.propertyName}${hasGuid || hasCancellat
ion ? ", " : ""} ${LINE_END}`;
    });
    if (hasGuid) result += `            cacheRevision${hasCancellation ?
", " : ""} ${LINE_END}`;
    if (hasCancellation) result += `            cancellationToken ${LINE_
END}`;
    result += "        );" + LINE_END;
    return result;
}
```

```
function addTypeLine(propertyName: string, typeName: TSType, isNullable:
boolean): string {
    const typingToAdd = `    ${propertyName}${isNullable ? "?" : ""}: ${t
ypeName};${LINE_END}`;
    return typingToAdd;
}
```

Příloha č. 20: Globální a lokální konfigurace

```
initializeConfig();
initSqlConfig();

const generateCommand = getGenerateCommand(yargs(process.argv.slice(2)));
const configCommand = getConfigCommand(generateCommand);
const listingCommands = getListingCommands(configCommand);
const regenerateCommand = getRegenrateCommand(listingCommands);
const argv = regenerateCommand.argv;

function getGenerateCommand(yargs: yargs.Argv): yargs.Argv {
    return yargs.command(
        "generate",
        "Generates data access",
        (yargs: yargs.Argv) => {
            return yargs
                .option("server", {
                    string: true,
                    description: "Specifies server name for future calls"
,
                    default: getConfigObject().global.server,
                })
                .option("database", {
                    string: true,
                    description: "Specifies database name for future call
s",
                    default: getConfigObject().global.database,
                })
                .option("schema", {
                    string: true,
                    description: "Specifies schema name for future calls"
,
                    default: getConfigObject().global.schema,
                })
                .option("callType", {
                    string: true,
                    description: "Specifies call type for future calls",
```

```
                    default: getConfigObject().global.callType,
                })
                .option("generateController", {
                    boolean: true,
                    description: "Specifies whether controller should be
generated name for future calls",
                    default: getConfigObject().global.generateController,
                })
                .option("generateDataAccess", {
                    boolean: true,
                    description: "Specifies whether data access should be
 generated name for future calls",
                    default: getConfigObject().global.generateDataAccess,
                })
                .option("generateModel", {
                    boolean: true,
                    description: "Specifies whether model should be gener
ated name for future calls",
                    default: getConfigObject().global.generateModel,
                })
                .option("generateInterface", {
                    boolean: true,
                    description: "Specifies whether interface should be g
enerated name for future calls",
                    default: getConfigObject().global.generateInterface,
                })
                .option("dataAccessPath", {
                    string: true,
                    description: "Specifies path to which generate data a
ccess name for future calls",
                    default: getConfigObject().global.dataAccessPath,
                })
                .option("controllerPath", {
                    string: true,
                    description: "Specifies path to which generate contro
ller name for future calls",
                    default: getConfigObject().global.controllerPath,
                })
                .alias("srv", "server")
                .alias("db", "database")
                .alias("s", "schema")
                .alias("ct", "callType")
                .alias("gc", "generateController")
                .alias("gda", "generateDataAccess")
                .alias("gm", "generateModel")
                .alias("gi", "generateInterface")
                .alias("dap", "dataAccessPath")
```

```
                .alias("cp", "controllerPath")

                .option("storedProcedureName", {
                    string: true,
                    description: "Specifies the of schema from which stor
ed procedures should be fetched",
                })
                .option("route", {
                    string: true,
                    description: "Specifies route of controller",
                })
                .option("httpMethodType", {
                    string: true,
                    description: "Specifies method type to use",
                })
                .alias("sp", "storedProcedureName")
                .alias("r", "route")
                .alias("http", "httpMethodType");
        },
        (argv) => {
            const commandArgs: CommandArguments = {
                server: argv.server,
                database: argv.database,
                schema: argv.schema,
                callType: argv.callType as MethodCallType,
                generateController: argv.generateController,
                generateDataAccess: argv.generateDataAccess,
                generateModel: argv.generateModel,
                generateInterface: argv.generateInterface,
                dataAccessPath: argv.dataAccessPath,
                controllerPath: argv.controllerPath,
                storedProcedureName: argv.storedProcedureName,
                route: argv.route,
                httpMethodType: argv.httpMethodType as "POST" | "GET",
            };
            generate(commandArgs);
        }
    );
}

function getConfigCommand(yargs: yargs.Argv): yargs.Argv {
    return yargs.command("config", "Used for setting or getting values of
 configuration file", (argv) => {
        return yargs
            .command(
                "set",
                "Used setting values in configuration file",
```

```
                (yargs) => {
                    return yargs
                        .option("server", {
                            string: true,
                            description: "Saves server name for future ca
lls",
                        })
                        .option("database", {
                            string: true,
                            description: "Saves database name for future
calls",
                        })
                        .option("schema", {
                            string: true,
                            description: "Saves schema name for future ca
lls",
                        })
                        .option("callType", {
                            string: true,
                            description: "Saves call type for future call
s",
                        })
                        .option("generateController", {
                            boolean: true,
                            description: "Saves whether controller should
 be generated name for future calls",
                        })
                        .option("generateDataAccess", {
                            boolean: true,
                            description: "Saves whether data access shoul
d be generated name for future calls",
                        })
                        .option("generateModel", {
                            boolean: true,
                            description: "Saves whether model should be g
enerated name for future calls",
                        })
                        .option("generateInterface", {
                            boolean: true,
                            description: "Saves whether interface should
be generated name for future calls",
                        })
                        .option("dataAccessPath", {
                            string: true,
                            description: "Saves path to which generate da
ta access name for future calls",
                        })
```

```
                        .option("controllerPath", {
                            string: true,
                            description: "Saves path to which generate co
ntroller name for future calls",
                        })

                        .alias("srv", "server")
                        .alias("db", "database")
                        .alias("s", "schema")
                        .alias("ct", "callType")
                        .alias("gc", "generateController")
                        .alias("gda", "generateDataAccess")
                        .alias("gm", "generateModel")
                        .alias("gi", "generateInterface")
                        .alias("dap", "dataAccessPath")
                        .alias("cp", "controllerPath");
                },
                (argv) => {
                    const config: GlobalConfiguration = {
                        server: argv.server,
                        database: argv.database,
                        schema: argv.schema,
                        callType: argv.callType as MethodCallType,
                        generateController: argv.generateController,
                        generateDataAccess: argv.generateDataAccess,
                        generateModel: argv.generateModel,
                        generateInterface: argv.generateInterface,
                        dataAccessPath: argv.dataAccessPath,
                        controllerPath: argv.controllerPath,
                    };
                    configureGlobal(config);
                }
            )
            .command(
                "get",
                "Used for getting values from configuration file",
                (yargs) => {
                    return yargs
                        .option("namespace", {
                            string: true,
                            description: "Defines namespace for the selec
ted configuration",
                        })
                        .alias("ns", "namespace");
                },
                (argv) => {
                    const globalConfig = getConfigObject();
```

```
                    if (argv.namespace) {
                        const config = globalConfig[argv.namespace];
                        if (!config) console.log(chalk.red("This configur
ation namespace does not exist!"));
                        else console.log(globalConfig[argv.namespace]);
                        process.exit();
                    }
                    console.log(globalConfig);
                    process.exit();
                }
            );
    });
}

function getListingCommands(yargs: yargs.Argv): yargs.Argv {
    return yargs
        .command(
            "listDb",
            "List available databases",
            (yargs: yargs.Argv) => yargs,
            (argv) =>
                listDatabases()
                    .then((res) => {
                        console.log(chalk.green("Resulting properties:"))
;
                        console.log(res);
                        process.exit();
                    })
                    .catch((err) => {
                        throw new Error(err);
                    })
        )
        .command(
            "listSchema",
            "List available schemas from a database",
            (yargs: yargs.Argv) => {
                return yargs
                    .option("database", {
                        string: true,
                        description: "Specifies the database from which s
chemas should be fetched",
                        default: getConfigObject().global.database,
                    })
                    .alias("db", "database");
            },
            (argv) =>
                listSchemas(argv.database)
```

```
                    .then((res) => {
                        console.log(chalk.green("Resulting properties:"))
;
                        console.log(res);
                        process.exit();
                    })
                    .catch((err) => {
                        throw new Error(err);
                    })
        )
        .command(
            "listSp",
            "List available stored procedures from a database",
            (yargs: yargs.Argv) => {
                return yargs
                    .option("database", {
                        string: true,
                        description: "Specifies the database from which s
chemas should be fetched",
                        default: getConfigObject().global.database,
                    })
                    .option("schema", {
                        string: true,
                        description: "Specifies the of schema from which
stored procedures should be fetched",
                        default: getConfigObject().global.schema,
                    })
                    .alias("db", "database")
                    .alias("s", "schema");
            },
            (argv) =>
                listStoredProcedures(argv.database, argv.schema)
                    .then((res) => {
                        console.log(chalk.green("Resulting properties:"))
;
                        console.log(res);
                        process.exit();
                    })
                    .catch((err) => {
                        throw new Error(err);
                    })
        )
        .command(
            "listSpInputs",
            "List available stored procedures from a database",
            (yargs: yargs.Argv) => {
                return yargs
```

```
                    .option("database", {
                        string: true,
                        description: "Specifies the database from which s
chemas should be fetched",
                        default: getConfigObject().global.database,
                    })
                    .option("schema", {
                        string: true,
                        description: "Specifies the of schema from which
stored procedures should be fetched",
                        default: getConfigObject().global.schema,
                    })
                    .option("storedProcedureName", {
                        string: true,
                        description: "Specifies the of schema from which
stored procedures should be fetched",
                    })
                    .alias("db", "database")
                    .alias("s", "schema")
                    .alias("sp", "storedProcedureName");
            },
            (argv) =>
                listProcedureInput(argv.database, argv.schema, argv.store
dProcedureName)
                    .then((res) => {
                        console.log(chalk.green("Resulting properties:"))
;
                        console.log(res);
                        process.exit();
                    })
                    .catch((err) => {
                        throw new Error(err);
                    })
        )
        .command(
            "listSpOutputs",
            "List available stored procedure outputs from a database",
            (yargs: yargs.Argv) => {
                return yargs
                    .option("database", {
                        string: true,
                        description: "Specifies the database from which s
chemas should be fetched",
                        default: getConfigObject().global.database,
                    })
                    .option("schema", {
                        string: true,
```

```
                        description: "Specifies the of schema from which
stored procedures should be fetched",
                        default: getConfigObject().global.schema,
                    })
                    .option("storedProcedureName", {
                        string: true,
                        description: "Specifies the of schema from which
stored procedures should be fetched",
                    })
                    .alias("db", "database")
                    .alias("s", "schema")
                    .alias("sp", "storedProcedureName");
            },
            (argv) =>
                listProcedureOutput(argv.database, argv.schema, argv.stor
edProcedureName)
                    .then((res) => {
                        console.log(chalk.green("Resulting properties:"))
;
                        console.log(res);
                        process.exit();
                    })
                    .catch((err) => {
                        throw new Error(err);
                    })
        );
}

function getRegenrateCommand(yargs: yargs.Argv): yargs.Argv {
    return yargs.command(
        "regenerate",
        "List ",
        (yargs: yargs.Argv) => {
            return yargs
                .option("storedProcedureName", {
                    string: true,
                    description: "Specifies already generated stored proc
edure",
                })
                .alias("sp", "storedProcedureName");
        },
        (argv) => {
            const globalConfig = getConfigObject();
            const currentConfig = globalConfig[argv.storedProcedureName];
            if (!currentConfig) {
                console.log(chalk.red("Configuration doesn't exist for sp
ecified procedure"));
```

```
                process.exit(1);
            }
            generate(currentConfig);
        }
    );
}
```

Příloha č. 21: tsconfig.json

```
{
    "compilerOptions": {
        "target": "es5",
        "module": "commonjs",
        "lib": ["es6", "es2015", "dom"],
        "declaration": true,
        "outDir": "lib",
        "rootDir": "./",
        "strict": true,
        "types": ["node"],
        "esModuleInterop": true,
        "resolveJsonModule": true
    },
    "exclude": ["result", "lib"]
}
```

Příloha č. 22: package.json

```
{
    "name": "dt-lgx-dafg",
    "version": "1.1.1",
    "description": "Logex Data Access File Generator for diploma thesis",
    "main": "index.ts",
    "bin": {
        "dt-lgx-dafg": "./lib/index.js"
    },
    "license": "ISC",
    "author": "Simon Prochazka",
    "repository": {
        "type": "git",
        "url": "https://github.com/SimProch/diploma-thesis"
    },
    "scripts": {
        "start": "node ./lib/index.js",
        "build": "tsc -p .",
```

```
        "listSp": "./examples/listSp.sh"
    },
    "dependencies": {
        "chalk": "^4.1.0",
        "figlet": "^1.5.0",
        "fs-extra": "^9.0.1",
        "msnodesqlv8": "^2.0.7",
        "mssql": "^6.2.3",
        "ts-node": "^9.0.0",
        "typescript": "^4.0.5",
        "yargs": "^16.1.0"
    },
    "devDependencies": {
        "@types/chalk": "^2.2.0",
        "@types/figlet": "^1.2.1",
        "@types/fs-extra": "^9.0.2",
        "@types/mssql": "^6.0.5",
        "@types/node": "^14.14.6",
        "@types/yargs": "^15.0.9"
    }
}
```